PENNSTATE

Final Report, December 2004

Applied Research Laboratory

MOBILE UBIQUITOUS SECURITY ENVIRONMENT (MUSE)

Critical Infrastructure Protection /University Research Initiative
Office of Naval Research (ONR)

Technical Contact:
Dr. Shashi Phoha
Director, Information Science and Technology Division
Assistant Director of Applied Research Laboratory and Professor of Electrical and Computer Engineering
Applied Research Laboratory
Penn State University
P.O. Box 30
State College, PA 16804
Telephone: (814)863-8005
Fax: (814)863-0679
Email: sxp26@psu.edu

Administrative Contact:
Mrs. Pamela Righter
Asst. to the Director for Finance, Business, and Administration
Applied Research Laboratory
Penn State University
P.O. Box 30
State College, PA 16804
Telephone: (814)863-3991
Fax: (814)865-2805
Email: prk3@psu.edu

Applied Research Laboratory
P.O. Box 30
State College, PA 16804-0030

Supported by the Office of Naval Research and Ralph Wachter, (the program manager) under Grant No. N00014-01-1-0859

EXECUTIVE SUMMARY

This is the final report of the Mobile Ubiquitous Security Environment (MUSE) Critical Infrastructure Protection University Research Initiative (CIP/URI) project. MUSE was proposed to research "understanding mobile code" in the context of Critical Infrastructure Protection (CIP). It made significant advances in this area. Mobile code differs from other software systems in that it uses networks to autonomously move code from one host to another.

Many common CIP threats, such as Trojan horses and viruses, pre-date widespread use of the Internet and are not specific to mobile code. Issues such as insuring program correctness, enforcing security policies, avoiding buffer overflows, and detecting malicious code also exist for non-networked software. Our emphasis is on researching how code migration affects infrastructure protection.

Viruses, worms, and Denial of Service (DoS) attacks are difficult to counteract in large part because they are highly distributed. Fortifying the defenses of individual processors, or even sub-nets, cannot sufficiently neutralize these threats. Our game theory analysis of DoS attacks contains examples of the limitations of firewalls for protecting distributed systems. Fortifying individual processors is in some ways similar to building a stronger Maginot line after World War II.

MUSE studied both the threat posed by malicious mobile code, and the promise of mobile code to adapt when attacked and neutralize threats. Distributed adaptation can put attacked systems on an equal footing with their attackers. The project Statement of Work (SoW) consisted of four tasks:

- Develop a theoretical model
- Study the interface between mobile code and the host computer.
- Study system adaptation
- Create an adaptive network infrastructure.

Significant results include:

- A theoretical model for mobile code was developed by integrating mobile agent and cellular automata concepts. Using a simulation tool (CANTOR), that we developed for this model, we found important behavioral differences among mobile code paradigms.
- CANTOR simulations were found to trend like other network simulators. The CANTOR models are simpler and contain fewer factors. They also execute more quickly than traditional approaches.
- A taxonomy of mobile code paradigms was created combining our theoretical model with an existing taxonomy of network attack vulnerabilities.
- Our existing mobile code daemons were integrated with peer-to-peer (P2P) indexing to create an initial adaptive infrastructure. Cryptographic key management has been integrated into this approach.
- We proved that cryptographic primitives can be used with a tamper-proof co-processor to verify bilateral trust between a host and a mobile code package. An application to multi-level security was given.

1. BACKGROUND

Code is mobile when it can change the location where it executes [Fuggetta 1998]. This differs from mobile computing, where hardware is displaced [Milojicic 1999]. Mobile code is particularly important as an adaptive framework for implementing network services. This requires the ability to find the global consequences of local behaviors of interacting automata. Ideally, we will find ways to engineer local behaviors that interact to fulfill global goal definitions. Many paradigms have been proposed as frameworks for mobile code. Most of these paradigms are unnecessarily restrictive. MUSE developed an abstract model that unifies known paradigms. The model was used to study mobile code at different levels of abstraction with twin goals: (1) establish the capabilities and limitations of mobile code use, and (2) produce technologies to protect critical national infrastructure from abuse of mobile code.

Mobility is basically a medium for delivering software to computers. As such it has global and local risks, which are not unique. Operating systems are vulnerable to viruses, Trojan horses, etc. [Tanenbaum 1997]. Computer networks are vulnerable to intrusion, denial of service, partitioning, etc. [Stallings 1995]. These security problems can disrupt operations, compromise sensitive information, and/or destroy data. The flexibility of mobile code combines the vulnerabilities of both computer operating environments and networks. This produces a negative synergy [Rubin 1998] with a potentially destabilizing influence.

Within the context of the program, a number of "disruptive technologies" were analyzed. These include mobile code, peer-to-peer networks and Field Programmable Gate Arrays (FPGA's). Each of these technologies is in the process of radically changing the use of computer networks. Each of these disruptive technologies produces new security threats. They also provide radically new capabilities for agile system adaptation. Our research indicates that this agility can be exploited for both securing and attacking networks. Many security aspects, both good and bad, of system adaptability have been ignored up to now.

2. STATEMENT OF PROGRAM RESULTS

The National Information Infrastructure (NII) has become a globally interconnected programming environment. Industry has discovered the natural utility of code and data migration within the NII. Transferring code and data automatically over the network is not *a priori* more dangerous than transfer by other means, but the use of mobile code amplifies security flaws in existing systems. This has resulted in well-known security breaches, such as the Internet worm, the Melissa virus, and the Love Bug. A truly robust NII requires advances in engineering complex adaptive systems based on mobile code. The MUSE program pursued a research agenda for in-depth modeling and understanding of code mobility. In particular, we:

- Created an abstract model for code mobility that established the local and global capabilities of mobile code, and its limitations.
- Created formal proofs of methods for establishing trust between mobile code and host computers.
- Established ways for computer programs to adapt to diverse machines that may be encountered on the NII.
- Created new technology for distributed adaptive services that compensate for failures and intrusions via mobility.
- Developed new technologies for secure processors, including creating FGPA based cryptography co-processors with performance superior to other documented implementations.
- Created models of network behavior used for intrusion detection.
- Performed strategic analysis of network security issues for Denial of Service (DoS) attacks and malware (virus and worm) propagation.

In conjunction, this work extended the realm of network security research into new areas. In particular, the "disruptive technologies" mentioned in the background section will cause increasingly large-scale changes in how software and hardware systems are designed, implemented and maintained. The security implications of these changes have been partially recognized [Rubin 1998, Oram 2001], but only as a threat to local processors. More extensive research is needed in the realm of global network security strategies, and autonomous adaptation to network attacks.

3. MODEL OF MOBILE CODE

This section discusses our abstract model of mobile code systems. The model has been used to analyze how local mobile code behaviors determine some aspects of global network behavior. We compare a network simulator created using our approach to a standard network simulation tool, and find that our results are comparable to results obtained using the traditional approach. Our approach requires much less time to execute. The model was also used to derive a taxonomy of mobile code systems, (which we use to illustrate that many aspects of network security related to the use of mobile code are being ignored.)

3.1 DESCRIPTION OF MODEL

In [Brooks 2002], we construct a model that expresses code mobility in a manner that includes established paradigms as special cases. The model uses Cellular Automata (CA) constructs. Computer nodes are elements in an extended CA. They exchange behaviors as needed. Environmental and other external influences are expressed as Free Agents in a Cellular Space (FACS). This allows us to model qualitative network attributes as a function of mobile code behavior. This is especially important when considering pathological network behavior caused by worms, viruses, etc.

The FACS approach used is inspired by literature on systems defined by interactions among multiple components in quantum optics, biology, and sociology. In biological systems, three modeling tools have been found useful: (*i*) differential equations, (*ii*) cellular automata, and (*iii*) multi-agent simulations [Camazine 2001]. Our approach combines concepts from cellular automata and multi-agent simulations into a common framework. For computer networks, differential equations may be problematic since data flows in discrete packets. When CA and differential equations model equivalent systems, the results are generally consistent. Differential equations results are of higher fidelity. On the other hand, numerical solution of differential equations using finite elements or finite differences methods is in many ways similar to creating CA approximations of differential equations.

Studies show internet traffic exhibiting a quasi-fractal nature with self-similarity over a wide range of time scales [Leland 1994, Grossglauer 1999]. Traditional queuing models do not adequately explain the burstiness of data flows [Willinger 1998]. The Internet is a decentralized system, whose global behavior is determined at a number of scales by interactions between a large number of autonomous systems and individual nodes. We consider network behavior an emergent system built of multiple interacting components. The global behavior of a network, like the Internet, is a function of both network configuration and behavior of the individual components.

Cellular automata models are powerful tools for studying large, interacting systems. Universal cellular automata can emulate arbitrary Turing Machines, guaranteeing that they are capable of executing any computable function. A CA is a synchronously interacting set of abstract machines (network nodes), defined by:

- d the dimension of the automata
- r the radius of an element of the automata
- δ the transition rule of the automata
- s the set of states of an element of the automata

An element's (node's) behavior is a function of its internal state and those of neighboring nodes as defined by δ.

Of primary interest is the design of systems that have globally desirable emergent behaviors, such as intrusion tolerance, based on locally available information. This can be approached in two ways: (*i*) the *forward problem* takes proposed local behaviors and determines their global consequences, (*ii*) the *backward* (or inverse) *problem* attempts to derive local behaviors with globally desirable attributes. The model we propose is a straightforward tool for evaluating the forward problem. The backward problem is still open.

The models we use build on CA based work for modeling traffic and socialsystems. This work involves CA environments where entities move from CA element to CA element. These traffic models are referred to as *particle-hopping models*. In [Portugali 2000], a similar concept is referred to as *Free Agents in a Cellular Space* (FACS). We use the FACS nomenclature.

Network traffic can be modeled by allowing each element to represent a computer node in a computer network. Packets move probabilistically from node to node. There is a maximum queue length. Our approach does this to model the flow of code, data, and coordination information.

In the context of mobile code, the behavior of the global network cannot be defined purely as an aggregation of individual node behaviors. The behavior of packets traversing the network is also important. In addition to this, mobile code modifies the behavior of its host node. We call our model *Interacting Automata Network* (IAN). CA's in the IAN model are non-uniform. They are defined by the tuple $<d,r,l[],\boldsymbol{\Delta},S[],B>$ where:

- *d* – dimension of the automaton
- *r* – radius of the automaton
- *l[d]* – vector indicating the number of elements in each dimension.
- *Δ* - set of transition functions. Each element could have a unique transition function.
- *S[]* – state vector for each automaton.
- *B* – set of behaviors. Behaviors are not tied to specific elements of the CA.

This model has the same essential form as a CA but is less simple. The goal is to use this model to study the possibility of network self-organization using mobile code. Study of these CA models should help establish the global behaviors implied by local mobile code activity.

Our approach to mobile code uses three abstractions: infrastructure, code, and data. There is a fixed infrastructure, which consists of computer hosts and network pipes connecting the hosts. This infrastructure is expressed primarily by the variables *d, l[d],* and *r* in the IAN tuple. In computer architecture the number of computer connections incident on a node is called the node degree, which is expressed by *d* in our model. The maximum distance between any two nodes is the network diameter, which is roughly equivalent to *l[d]* in the proposed approach. In most models we use *r* equal to one, but it could be set to a larger value to model bus architectures. In addition to *d, l[d],* and *r,* individual transition functions in *Δ* are used to complete the description of network topology by either allowing or prohibiting communications between IAN elements.

A three-dimensional topology is appropriate for modeling wireless communications networks. It is also possible to construct a model for an arbitrary network. If there are *n* nodes in the network, the dimension *d* can be set to *n* and all elements of *l[d]* set to one. This implies each node is directly connected to every other node. Judicious definition of the elements of *Δ* prohibits communications between pairs of nodes that are not directly connected. Using this approach, any arbitrary graph structure of *n* nodes can be embedded in an *n* dimensional grid.

Each element of the IAN has its own instance of the state vector *S[]*. Having the state be a vector is a departure from the traditional CA model, where state is typically a single discrete variable. Use of a vector is needed to concurrently capture multiple aspects of the problem space. For example, behavior of a network will generally depend on the volume of data being handled by the network and the queue length at specific nodes. CA based particle hopping models provide qualitative support for modeling these types of networks. In addition to this, we wish to model the behavior of networks influenced by mobile code.

The presence (or lack) of specific mobile code modules influences the behavior of network nodes as well. Another factor that influences node behavior is whether or not intrusions have been detected. Although it would be possible to use coding schemes to integrate most, if not all, of these factors into a single value, that would serve to complicate rather than simplify the approach.

We force the format of *S[]* to be uniform for all elements of the IAN. Constricting the elements of the system to a common state format is standard. The main reason why all elements need a common state definition is to support analysis of the IAN evolution. For example, visualization of queue length changes across the network illustrates network congestion. Visualization of the diffusion of mobile code, or network viruses, across the network helps understand the ability of the code to modify network behavior. We also want to visualize the distribution of entropy throughout the network. All of these tasks imply a uniform basis for evaluating state for all elements in the IAN.

The behavior of a specific IAN element is defined by the interaction between the *S[], B,* and *Δ* local to that node. In principle, all elements evaluate their δ concurrently taking as inputs current values of the state vectors and concurrently producing new state vector values for all elements of the IAN. At each time step, each element of the IAN evaluates its own transition function δ, which is an element of *Δ*. Based on the values in the local state vector and the state vectors of all the neighbors, δ determines new values for the local *S[]*. To a large extent, δ is defined by the subset of *B* present on the local node. Since behaviors are mobile and can move from node to node, δ can vary over time. Elements of *B* are arbitrary programs that may have their own state. When executing, these programs only have access to information and resources on the local node.

The migration of a behavior from node *j+1* to node *j* at time step *t* can be illustrated representing the local transition function as a finite state machine, which is the parse tree generated from the set of behaviors local to *j*. At time step *t*, the behavior attached to the parse tree on node *j+1* at position *x* moves to node *j*. This produces a new parse tree. The migration of this behavior has modified node *j*'s transition function. We call each time step a *generation*. During a generation, each node computes the state it will have at the next generation. This simulates the parallelism inherent in networks. Adding probabilities to interactions between nodes, allows simulations to act as if each node had its own clock.

Individual elements can also have memory storage beyond a finite number of states and model a Turing or von Neumann machine. Our modifications to the normal CA model reflect the reality of distributed systems for viruses, network intrusions and mobile code applications. Using it, we can model intrusion propagation through a network, and network reconfiguration to battle intrusion. All actions are taken by individual nodes based on *S[]* and δ; using locally available information.

To use this approach, we constructed a simulator based on the IAN model. The simulator consists of a front end that emulates a network with mobile code, and a back end that can be used to visualize the evolution of the IAN. We have named the tool CANTOR (CA ANimaTOR) in honor of the mathematician Cantor's work on self-similar sets. We use this tool to analyze multiple instances of the forward problem for designing robust networks safeguarded by mobile code behaviors. CANTOR users can do the following:

- Construct *k*-dimensional cellular grids
- Construct rule based cellular neighborhoods
- Assign evolutionary rules on a cell-by-cell, region-by-region, or grid basis
- Record evolution of cell states
- Model free agents in the cellular space
- Construct re-writable, swappable and evolving rules
- Visualize the results in 2 or 3 dimensions
- Reprogram the system using a scripting language
- Perform mathematical analysis of cellular grids.

CANTOR was designed to model network traffic in distributed systems, but has evolved to be capable of modeling dynamic discrete event systems using cellular automata. Data analysis tools include, entropy analysis, probability and statistical summaries of generations and runs.

3.2 SIMULATIONS

In this section we show how mobile code paradigms can be phrased as examples of our model. The model we propose is thus more inclusive. This approach has many pragmatic aspects. Using abstract automata allows us to quickly construct models of a specific problem for simulation and analysis. The CA shows interactions among distributed components. The tools in section II allow a quick evaluation and comparison of different approaches to a given problem. The tuple elements *d*, *r*, and *l[d]* define network topologies. A network instance that a paradigm will be used in will have a particular topology. The mobile code paradigm itself is described in terms of *S[]*, *B*, and Δ.

3.2.1. CLIENT/SERVER

In the client/server paradigm, one computer node has a program that one or more remote computers need to execute. For example, a single database server can service the needs of multiple hosts. A common implementation of client/server is a three-tier approach.

A number of clients (layer 1) send requests to a well-known address (layer 2) that feeds them to a central server (layer 3). The central server is provided with a client-specific address (layer 2) for forwarding results to the client. It would be possible to model this in a number of ways. Here we combine layers 2 and 3 in a single layer. We provide an example. Since this approach is independent of the network topology, *d*, *r* and *l[d]* can be arbitrary.

One IAN element is specified as the server. It handles the duties of the second and third (*S*) layers. It would be possible to have a separate second layer. The rest of the elements in the IAN would be clients (*C*). The server cannot be a client.

We define *S[]* as a vector with the following members:

- *Role* – a nominal variable identifying the IAN element as *C or S.*
- *Packet queue length* – the number of packets currently on the node awaiting transmission to a neighboring node. (all elements)
- *Packet list* – the store-and-forward packets are kept in this FIFO temporary data storage. Each packet would be a data structure containing source, destination, and request specific information. (all elements)
- *Outstanding requests* – the number of requests waiting for a response from the server. (all)
- *Serviced requests* – the number of requests that have already been serviced by the server. (all)

- *Request list* –list of requests that have yet to be serviced (second tier and server).
- *Internal state* – each node needs a set of states (ex. waiting for request, processing request, etc.) for interaction with its transition function δ. (all nodes).

State elements that are not relevant for a given IAN type are set to a known default value. Two different transition functions exist in this model:

- *Client* – Starts in an idle state. When a request is made, it triggers a request behavior that propagates through the network and transitions to a waiting for response state. While in that state, it can make further requests. Each time a request is made the number of outstanding requests is augmented. When a response to a request is received, the number of outstanding requests is decremented and the number of serviced requests incremented. If the number of outstanding requests reaches zero, the client returns to an idle state.
- *Server* – Starts in an idle state. As requests are received, the original request behavior terminates. Outstanding and serviced request counters are maintained. It moves to a waiting for response state when a service request is received. When requests are serviced, it moves to a responding to service state. In this state, it terminates the results behavior and starts a new one. In addition to this, the outstanding requests queue is maintained.

In these definitions, we specified neither how clients formulate requests nor how the server formulates responses.

Most of these transition functions are essentially First-In First-Out (FIFO) queues. They could be implemented either as a finite state machine, or as a stack machine. For the finite state machine, queue length relates directly to the state of the automaton. The maximum queue length is exceeded when a request is received and the machine is in state n. For the stack machine, queue length corresponds to stack depth. The stack machine abstraction does not provide any advantages over finite state machines for expressing limited queue lengths.

To analyze client/server performance versus other designs: (i) clients produce requests with a Poisson probability distribution, and (ii) the server replies to requests in a FIFO manner following an exponential probability distribution.

Another issue we consider is packet transmission over the network. There are two ways of modeling this. It can be either expressed in the transition function, or as mobile behaviors. It is worth noting that it is trivial to insert elements into the IAN, which are not directly involved in the client-server mechanism and only relay packets between neighboring elements. To express packet transmission in the transition function, each element is made responsible for exchanging packets with its neighbors. Packets have source and destination addresses. Per default, packets originate at their source and travel to their destination following a simple greedy heuristic. They move from their local position to the next node, which has not reached its maximum queue length, along the shortest path to the destination. The default process is to use the particle-hopping model. At each time step, the packet at the bottom of the FIFO packet queue is potentially transferred to its neighbor. Only one packet is transferred per time step in all simulations presented in this paper. Transfer of packets follows the general form of the Nagel-Schreckenberg particle-hopping model. This simple behavior is present in every node and can be expressed as a finite state machine or stack machine. By performing a cross product operation between this finite state machine and the transition functions described above, the two functionalities can be merged in a straightforward manner. In the simulations we present here, packets are transmitted in this manner unless otherwise indicated.

If more complex routing behavior is desired for packets, they can also be expressed as behaviors. In this client-server model, we have defined behaviors for request and results propagation. These behaviors originate at the sending IAN element. They execute once on each element they visit. The execution occurs when they are at the bottom of the FIFO queue of packages. Their logic consists of finding the appropriate next hop on their way to their destination; creating a copy of themselves on the next hop IAN element; and removing themselves from the current IAN element. Their tasks can include bookkeeping on current and neighboring IAN elements. In addition to source and destination addresses, the behaviors also include data elements specific to the service requests.

For all images showing the evolution of CA models, we will use the same axes. Each pixel along the x-axis is a client. The y-axis shows how the system evolves over time. No messages go beyond the boundaries, since messages are exchanged only between nodes active in the simulation. Nodes are numbered left to right from 0 to n. Messages for lower (higher) numbered nodes go left (right).

Figure 1a shows the graphical results of a CANTOR client/server network simulation with a long mean time between requests by clients each following a Poisson distribution and low mean time to process the requests following an exponential distribution at the server. For this case, the central pixel marked in green is the server. Gray indicates an empty network queue. Red indicates an occupied network queue. Lighter shades of red show a longer queue. Note the transient congestion forming in the network. This is an example of data bursts forming spontaneously in the system. Except where indicated differently, the parameters for simulations are:

- Mean time between requests 75 generations.
- Mean processing time 1 generation.
- Maximum queue length 14.
- Retransmits enabled.
- Probability of transmission failure 0.05.
- No packet dropping.
- 300 generations.
- One-dimensional network of 200 nodes.

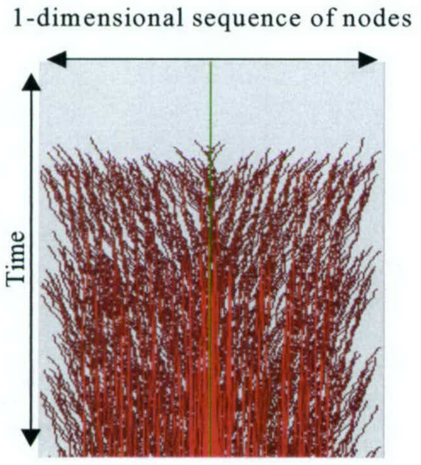


Figure 1a: Low arrival rate, high processing speed

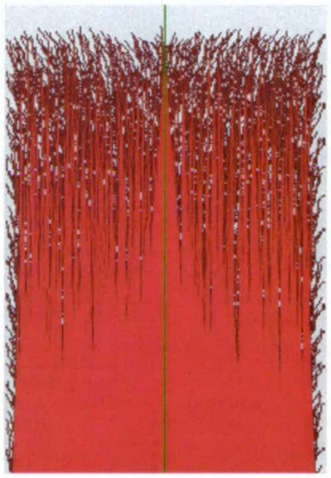


Figure 1b: Higher arrival rate, lower processing speed.

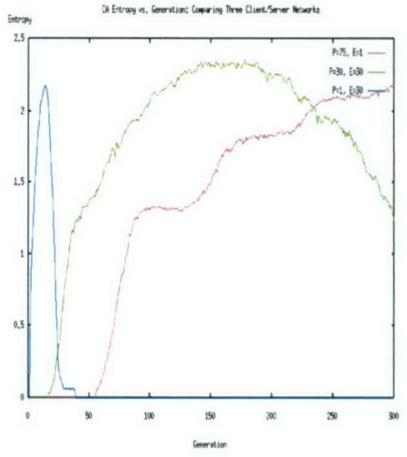


Figure 1c: Entropy plot for representative client/server simulations.

Figure 1b shows a network where the mean time between requests is much less (25 generations) and the mean time required to process requests is longer (3 generations). The results are consistent with what one would expect from a Queuing Theory analysis, significant congestion forms around the server.

Often entropy is used to analyze cellular automata and identify disordered regions. We find entropy charts of networks help identify bifurcations of network behavior. If x_i^t is the queue length of cell i at generation t, then the maximum likelihood estimator for the probability of a cell having queue length q at generation t is:

$$p_q^t = \frac{1}{N}\sum_{i=1}^{N} \chi\left(x_i^t\right)$$

where,

$$\chi\left(x_i^t\right) = \begin{cases} 1 & x_i^t = q \\ 0 & \text{else} \end{cases}$$

and N is the number of cells in the model. The entropy for generation t can be computed as:

$$H(t) = -\sum_{0 \le q \le q_{\max}} p_q^t \log\left(p_q^t\right)$$

Figure 1c shows the entropy for three client/server examples.

In the legend of **figure 1c**, P indicates the Poisson mean time between transmissions, and E indicates the mean time to service a request by the server. In this example, we observe three qualitatively different regimes:

- When the server is slow a long queue forms quickly around the server and congestion from this queue saturates the network (**figure 1b**).
- When the server is fast and jobs are generated quickly, the congestion shown is caused by network traffic and is distributed throughout the system (**figure 1a**).
- When the server is fast and jobs are generated slowly, there is little congestion. Queue formation is transient.

Similar effects are observed in other mobile code paradigms, which will be discussed in the following sections. For all cases if jobs are generated more quickly than the server can process them, the final state will be a deadlocked system with full queues.

3.2.2. REMOTE EVALUATION

The remote evaluation paradigm is similar to client/server, except the remote node may have to download code before execution. A common implementation of this is the use of remote procedure calls (RPC) on UNIX systems. The model given here builds upon the client/server model, with three modifications: (i) there is no second layer, (ii) the system is not limited to a single server, and (iii) the program executed is not generally resident on the server beforehand.

The packet transfer model can be taken from the client-server model with no modification. The client model requires little modification. The difference is that there is more than one server. Clients can make calls to any available server.

Similarly, servers work as with client-server with a minor modification. The modification is that if the behavior requested is not present it must be downloaded from another node. The request for code could be handled using a client-server approach. For the sake of simplicity, we do not designate a single server; we treat all nodes as peers. In which case, if a client node *a* requests execution of a program *p* on server node *b*, server node *b* may have to play the role of a client when requesting delivery of *p* from another node *c*. Upon completion of execution *b* returns results to *a*.

In the simplest model for remote evaluation, all elements of IAN play the role of client or server as needed. The state vector *S[]* is given by:

- *Packet queue length* – as in section 3.2.1.
- *Packet list* – as in section 3.2.1.
- *Client outstanding requests* – the number of requests waiting for a response from a server.
- *Client serviced requests* – the number of requests that have already been satisfied by a server.
- *Server outstanding requests* – the number of requests waiting to be serviced by this node.
- *Server serviced requests* – the number of requests that have already been serviced by this node.
- *Request list* – list of requests that have yet to be serviced by this node.
- *Internal state* – each node needs a set of states (ex. waiting for request, processing request, etc.) for interaction with its transition function δ.

This state structure reflects the fact that each node is both client and server.

In this approach there is only one transition function. Each node starts in an idle state. When necessary it performs client processing as described in 3.2.1 If the node receives a request, it performs the server processing as in 3.2.1. If the code requested is not present, it performs client processing to retrieve the code before servicing the request. To allow concurrent client and server processing, a cross product is formed of the client and server finite state automata in 3.2.1.

Simulation of this approach by a IAN implementation is straightforward. IAN elements are chosen at random to play the role of client or server. Specific network configurations can be implemented as needed. Deterministic schemes can then be used to test specific load models. Remote evaluation was an evolutionary extension to client server in that the code evaluated no longer had to be present on the server beforehand.

Figure 2 shows simulations of remote evaluation. (**Figure 2a -** Mean time between requests 30 generations). (**Figure 2b -** Mean time between requests 15 generations. Mean processing time 3 generations). Entropy plots for remote evaluation (not shown) are very similar to client server. The difference is that there is not a central server and processing is spread throughout the network. Instead of having a central bottleneck, a number of smaller bottlenecks occur.

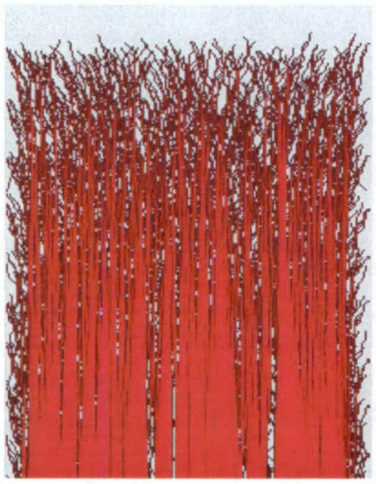


Figure 2a: Remote evaluation without significant congestion.

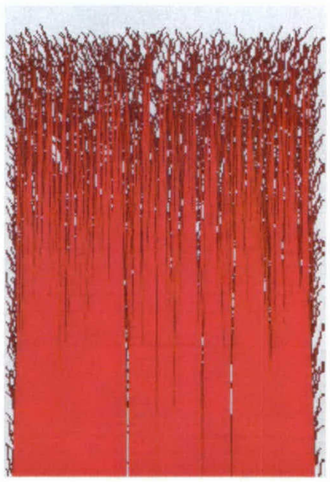


Figure 2b: Remote evaluation with significant congestion.

3.2.3. Code-on-demand

A common implementation of code-on-demand is a web browser. It is a simplification of remote evaluation. The server and client nodes are identical. A single user node can request code and data from any machine on the network. In some ways, it is the inverse of the client server model.

Code-on-demand can be phrased in IAN by modifying remote evaluation. Each node potentially contains a web browser. It can request code or data from other nodes on the network. The nodes respond by transmitting the information to the requesting node.

States and transition functions are the same as for remote evaluation. The differences are:

- Code always executes on the node that originates the request.
- Since results are computed at the node requesting the data, results do not have to be transmitted after computation has completed. Every node can potentially place a request. The number of nodes that can provide code and/or data to satisfy the request can be limited.

Code-on-demand extended remote evaluation by allowing arbitrary nodes on the network to request and execute code. The IAN models for code-on-demand show a marked difference in qualitative behavior with the client/server model. **Figure 3a** (Mean processing time is not applicable to code-on-demand) shows results from a code-on-demand network with a high Poisson mean time to generate requests. **Figure 3b** (Mean time between requests 15 generations) shows the same network with a low Poisson mean time to generate requests.

Figures 3c and **4** compare the three paradigms (30 generations mean time between request arrival for all three and processing time of 30 generations for requests where appropriate). Differences in the entropy plots of the three scenarios are evident. As the figure indicates, the disorder produced in the Code-on-Demand network seems to be greater than that of the other two networks. In both client/server and remote evaluation, computation requests are serviced remotely. The server has a queue that is serviced following an exponential distribution. In the code-on-demand model, requests are serviced by sending the program instantaneously. The delay introduced by the wait at the remote node appears to reduce disorder in the network. For code-on-demand, the behavior of the global system is dependent only on the network traffic. Processing is done locally and the time required to process requests is of no consequence to the behavior of the network as a whole.

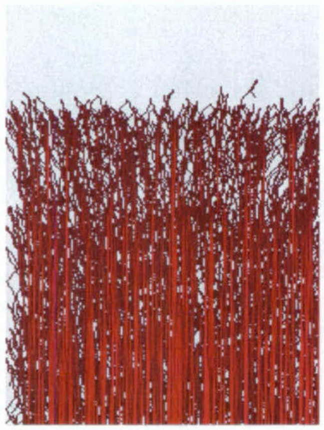


Figure 3a: Code-on-demand with minimal congestion.

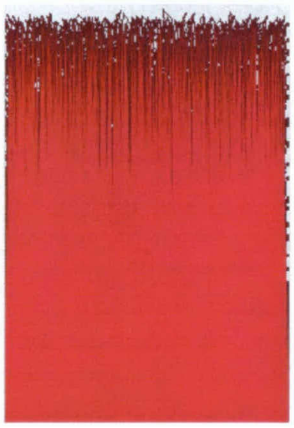


Figure 3b: Code-on-demand with significant congestion.

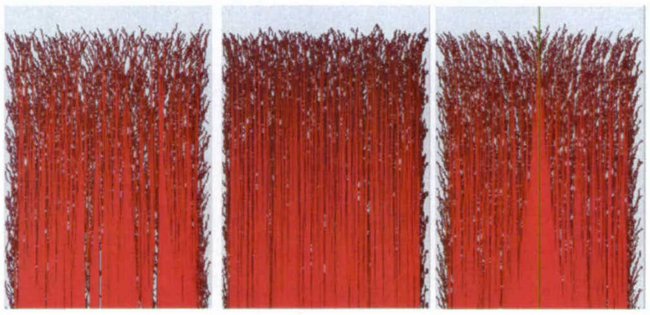


Figure 3c: Remote evaluation, code-on-demand, and client server with moderate congestion

Disorder is a function of queue length variance in the system. In practice, queue lengths are uniform under two conditions: (*i*) if the system is underutilized queue lengths are uniformly low, (*ii*) if the system is overloaded queue lengths are uniformly high as the system becomes congested. This implies that high entropy is indicative of an active system with transient congestion and no major bottlenecks. Using this argument, **figure 5** implies that code-on-demand is a more efficient approach. In this simple example all packets are of the same size. Code-on-demand generates significantly less traffic than the other two approaches. Since all processing is done locally, no response packets are generated. More surprising is the similarity between remote evaluation and client server. Remote evaluation has no central server bottleneck. It has 200 nodes acting as servers. One would expect the throughput of remote evaluation to be much higher. This does not appear to be the case. Evidently in this configuration, the congestion bottleneck is network traffic rather than server throughput.

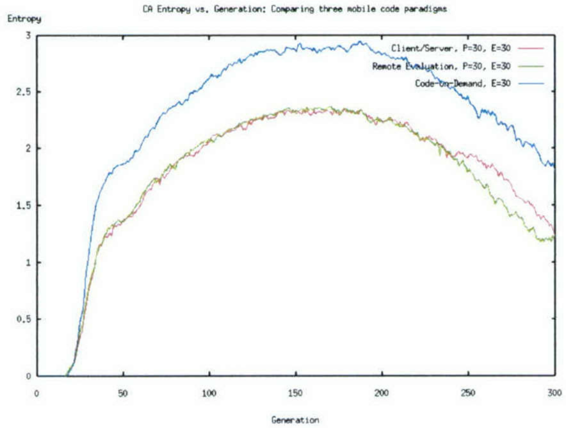


Figure 4: Entropy for remote evaluation, client server and code-on-demand.

3.2.4. PROCESS MIGRATION

Sections 3.2.1-3.2.3 describe widely used mobile code paradigms, which incrementally increase user independence from physical location. In contrast to this, process migration has been implemented and used primarily by research laboratories [14]. It differs greatly from the methods in sections 3.2.1-3.2.3. Among other things, it uses strong mobility to transfer program state along with code and data.

Processes move from node to node in an attempt to find an equitable distribution of resources. Process migration is in many ways a precursor of the mobile agent paradigm, which will be described in section 3.2.5

Since processes are not tied to their host nodes, we model them as instances of behaviors. Process behaviors are mobile. They include: (*i*) the program being executed, (*ii*) its state information, and (*iii*) associated data files. A process looks at the load on the local computer node (element) and all nodes (elements) in the immediate neighborhood. If the current load exceeds a threshold value and there exists a neighboring node with a lighter load, it moves to the neighbor with the lightest load. Processes may be large; transporting a process from one node to another may require multiple time steps.

Each node has a state vector *S[]* with the following members:

- *Outgoing packet list* – If a process is large, it must be cut into multiple packets for transmission from a node to its neighbor. Multiple processes may have to migrate simultaneously. All packets awaiting transmission are stored on a common FIFO queue.
- *Outgoing packet queue length* – Number of packets currently on the queue.
- *Incoming packet list* – This list temporarily stores packets that make up a process until it is entirely transferred.
- *Incoming packet queue length* – Number of packets currently on the list.
- *Maximum load* – an integer signifying the resources available at this node for use by processes.
- *Current load* – resources currently consumed by processes on the node.
- *Threshold load* – number of processes that can be active without processes migrating.
- *Process list* – a list of behaviors currently active on the node.

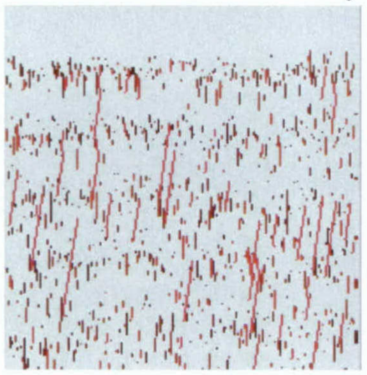


Figure 5a: Process migration, low process creation rate.

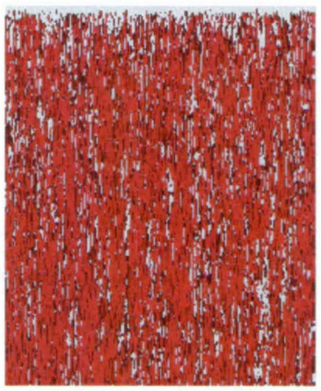


Figure 5b: Process migration, high process creation rate.

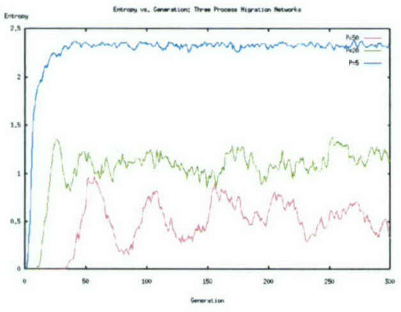


Figure 5c: Entropy for process migration

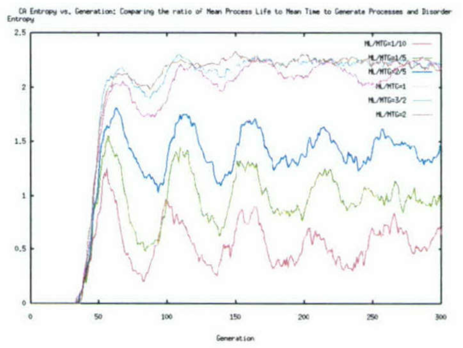


Figure 5d: Entropy shape as a function of life-time to production time

Each node has the same transition function, which resembles the Nagel-Schreckenberg particle-hopping model. Each process is chopped into a sequence of packets and put on a queue. Packets on the outgoing queue are transferred to the neighboring node chosen by the process. Once all packets associated with a process have been completely transferred to the neighboring node, the migrated process is put on the process list. One packet per time step can be transferred. There is a small probability that a packet will need to be re-transmitted.

Process behaviors follow a simple logic. When they start on a node they add their load to the current load. When they leave a node, they remove their load from the current load. They migrate to the neighbor with the smallest load. **Figures 5a** (Mean process load of 1 (exponential distribution), Maximum load supported by a node 5, Mean process duration 10 generations (exponential distribution), Mean time between process initiation 80 (Poisson)), and **6b** (Mean process load 1 (exponential distribution), Mean process duration 10 generations (exponential distribution), Mean time between process initiation 10 (Poisson)) show scenarios with high and low process arrival rates. Re-transmission was not included in the process migration and mobile agent scenarios.

Note that instead of showing packet queue length, **figures 5a & 5b** show the number of processes active on each node. The network disorder caused by process migration differs from the other networks we have modeled as well. **Figure 5c** shows the entropy plots of three process migration networks.

The blue line shows the entropy when the mean time between process arrivals following a Poisson distribution is five time steps. The system reaches a high state of disorder and remains at a constant high entropy. The green line in the middle shows a mean time parameter of twenty time steps. The entropy is lower, and it varies significantly over time. The bottom line shows a system with a mean process arrival time of fifty. The entropy is lower and shows a significant, regular oscillation.

We have analyzed the cause of this oscillation and determined that the period of oscillation is a function of the mean time between process generation, while the amplitude is a function of the ratio of mean process life-time to mean time between process creation. **Figure 5d** shows a set of entropy curves in which the mean time between process generations is fixed at 50 generations and the mean process lifetime is varied from 5 generations to 100 generations. As the figure shows, the period of oscillation is related to our use of a mean time between process generation of 50 generations. We see that peaks seem to be forming around the generations of multiples of 50. The amplitude of oscillation decreases as a function of the ration of mean lifetime to mean time between generation of new processes. In our opinion, the oscillation is a simulation artifact due to a correlation between these two factors. It is only significant at the start of the simulation and dies out over time. As the network becomes cluttered the peaks become less obvious.

3.2.5. MOBILE AGENTS

We define mobile agents as processes capable of moving from node to node following their own internal control logic. As such, mobile agents are an extension of the model proposed for load balancing. The difference being that their control logic is not limited to moving to neighboring nodes in response to load increases. In principle, this model could encapsulate arbitrary control logic. In our model, they can move to arbitrary nodes, but must pass through all intermediate nodes along the way.

The mobile agent approach has the constraint that data sources are generally not mobile. When agents move they transport only the data that is part of their internal state. They are often proposed as models where they harvest data from large distributed data resources.

The state vector *S[]* is the same for mobile agents as for process migration, except that the load variables are removed. The transition functions for nodes (elements) are identical to the process migration transition functions.

Agent behaviors can potentially follow arbitrary logic. In our model, few restrictions exist on the internal structure of the behaviors implementing the mobile agents. In our analysis, an agent is given an arbitrary itinerary. The itinerary consists of a list of nodes to be visited and processing to be done. The processing times vary according to a uniform distribution. **Figures 6a** (Mean process load 1 (exponential distribution), Mean process duration on each node 8 generations (exponential distribution), Mean time between process initiation 50 (Poisson)) and **6b** (Mean process load 1 (exponential distribution), Mean process duration on each node 8 generations (exponential distribution), Mean time between process initiation 10 (Poisson)) shows the results of a IAN simulation of a network employing mobile agents. These diagrams show process queue length. Note that the generality of the mobile agent concept means that other types of mobile agents are possible.

Figure 6c shows the entropy plots for representative mobile agent networks. The entropy plot resembles the entropy of process migration systems in many respects. For a large number of agents, this similarity is reasonable.

For a medium and small number of agents, the entropy is higher for the agent system than for process migration. This is reasonable since the system is not attempting load balancing.

We can see a similar oscillation in the mobile agents. We believe this oscillation is caused for the same reasons as in the process migration model and is a simulation artifact.

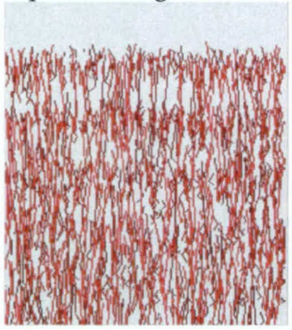


Figure 6a: Mobile agent network with low congestion

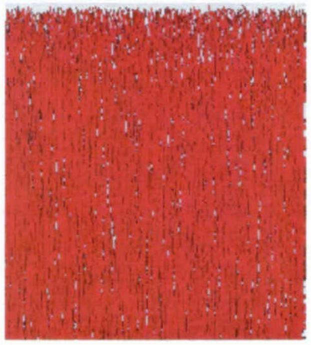


Figure 6b: Mobile agent network with high congestion

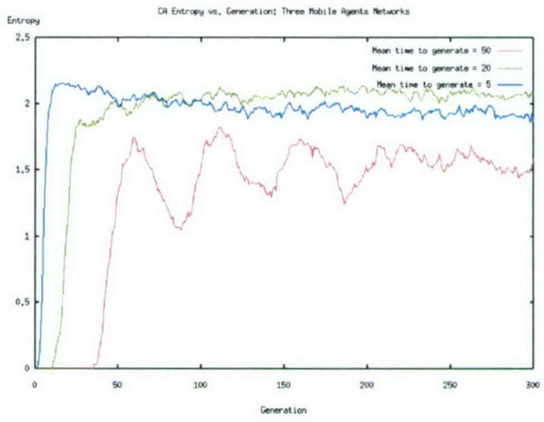


Figure 6c: Entropy plots for mobile agent networks

3.2.6. WORM

A worm is a network program that creates new copies of itself on a computer system. Worms most frequently attempt to create multiple copies of themselves on multiple computers. We consider worms as mobile agents that not only move from node to node, but also create new instances of themselves. **Figures 7a** and **7b** (mean time between process initiation 15 generations) show some sample runs of a network that has been infected with a worm. **Figure 7c** shows the entropies of other representative networks that have been infected with worms. The reader will note the increased traffic that results from a worm versus a regular mobile agent. Note that worms are not the only mobile agent variations that can produce new copies of themselves.

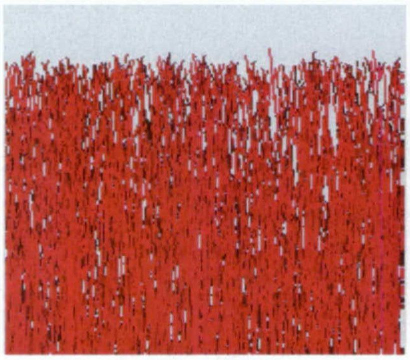


Figure 7a: Slow spread worm infection

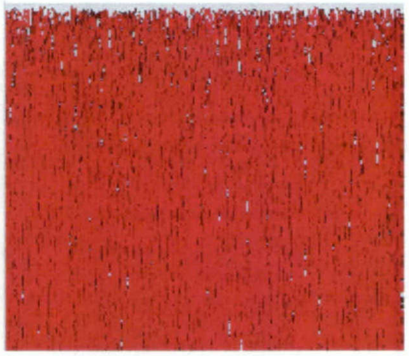


Figure 7b: Fast spreading worm infection

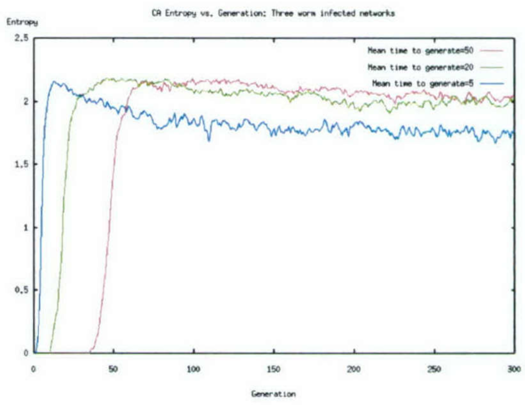


Figure 7c: Entropy plots for worm infected networks

3.2.7. VIRUS

A virus is a program that lives within another program. When the host program runs, the virus logic is activated first. The virus logic searches for another program to infect; and copies itself into that program. This can best be modeled as an instance of an active network or mobile agent. The logic carried in a packet modifies the host node by inserting into the host node logic that infects packets as they pass through the host node.

3.2.8. DISTRIBUTED DENIAL OF SERVICE

A denial of service attack is an instance of the client server paradigm. A client floods a server with requests until the server can longer function. Distributed denial of service (DDOS) attacks are examples of the remote evaluation paradigm. A single client tasks multiple nodes on the network to flood a given victim node with requests until the victim can no longer function.

Figures 8a (10 percent of the nodes are zombies, for the other nodes: mean time between request initiation 40 (Poisson), mean time to service request 3 (exponential)) and **8b** (25 percent of the nodes are zombies, for the other nodes: mean time between request initiation 40 (Poisson), mean time to service request 3 (exponential)) show two examples of DDOS attacks. The green node is the server in these images. Yellow nodes represent zombie processes that attempt to cause congestion around the server. Each zombie generates a request for service from the server at each time step. In both images, congestion forms quickly around the server. Congestion can be seen moving quickly towards the server. This attack exploits the congestion weakness of the client server paradigm.

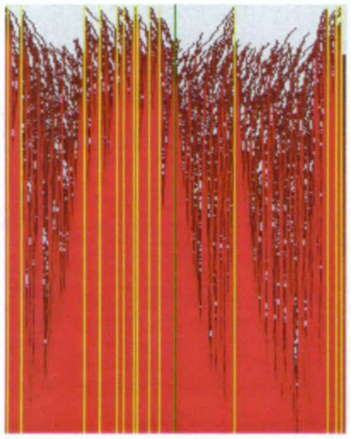


Figure 8a: DDoS attack, low number of zombies

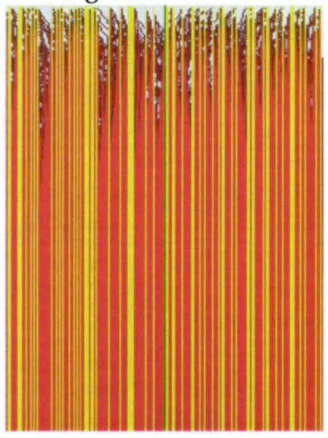


Figure 8b: DDoS attack, high number of zombies

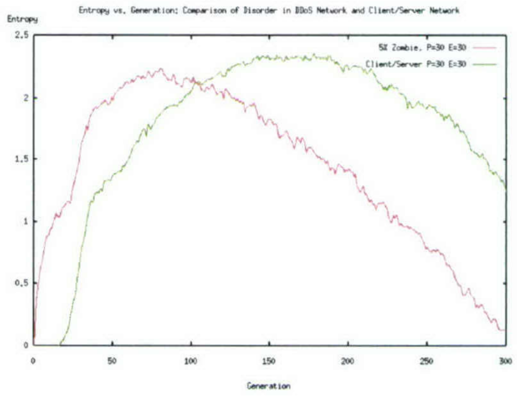


Figure 8c: DDoS entropy analysis

The entropy plot shown in **figure 8c** illustrates that the entropy profile of a DDOS attack is qualitatively similar to a client server entropy plot. The difference is that the DDOS system becomes disordered more quickly and that entropy dies off more quickly as congestion isolates the server from its clients.

3.2.9. COMMENTS ON MOBILE CODE MODEL

Networks built on mobile code systems have global characteristics. These characteristics are due to interactions among individual components. The approach provided here combines concepts from cellular automata and multi-agent simulations into a common framework.

In many ways, our approach builds on traditional queuing theory. The asymptotic results for the client server model are what one would generally expect from a queuing theory model. If the rate a server processes jobs is faster than the arrival rate; congestion is not significant. If that is not the case, congestion is significant. It is currently difficult for queuing theory to explain the behavior of the Internet. The probability distributions used for telephone networks do not fit the data. The proper long tailed distributions have not been found. Some theorize that appropriate distributions may have properties such as infinite mean and variance [9]. Current explanations for data burstiness, long-range network traffic dependence and multiple time scale self-similarity are also inadequate.

In some ways, our model is closer to the reality of computer networks than queuing theory based on Markov chains. Data traverses the system in discrete quanta. It is possible to embed complex decisions into components. We can model the flow of information to network components for making decisions. The models are still abstractions and not overly complicated by implementation details. With appropriate modifications this model may be able to yield quantitative, predictive results.

The model given here provides a simple abstract description of component actions in a mobile code system. Many complex qualitative phenomena can be observed, including data burstiness. At a qualitative level, the results here are consistent with behaviors found in the Internet. If we succeed in producing simple models with traffic statistics like the Internet, this will help determine the factors dominating the Internet's global behavior. Entropy measures appear to be useful for differentiating between different behavior regimes in the global system.

The use of simulations and visualizations is essential to this work. It is difficult to understand behaviors due to interactions among multiple components. Use of a CA to show system evolution over time helps students and researchers more easily understand network congestion patterns. The influence of random factors and transients in the network can be analyzed by viewing multiple runs using different seeds and initial conditions. In designing these models care should be taken to include only essential factors. It may only be possible to determine these factors through experimentation.

3.3 COMPARISON OF CANTOR TO OTHER NETWORK SIMULATIONS

To verify the utility of our CA model, we constructed CANTOR models of the TCP and UDP IP transport protocols. These models are given as algorithms 1, 2, and 3 on the following pages. Example networks were constructed and the dynamics of our CA based transport models compared with results from high fidelity network simulators.

Figures 9 and 10 show results from the UDP and TCP tests respectively. The CANTOR UDP results are virtually identical with the ns-2 results, with the exception of a small offset. The CANTOR TCP results trend similarly to the ns-2 results. That deviations exist between ns-2 and our simulation is not surprising, since we implemented a minimal TCP model. The payoff of the minimal model is shown in figure 11, which shows that our technique scales much better than traditional network simulations.

Figure 12 shows the new CANTOR visualization interface that allows us to display system interactions for very complex network scenarios.

Algorithm 1 UDPUPDATE: UDP update rule

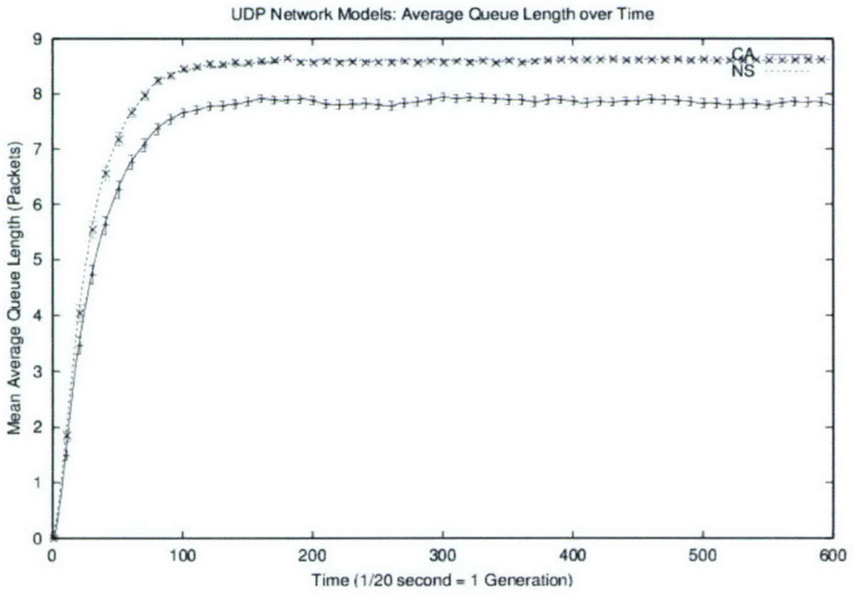
```
Input: Cell State STATE, Neighbors NEIGHBORS;
Output: None;
 1: {Check neighbors for new mail packets}
 2: for all NEIGHBOR $\in$ NEIGHBORS do
 3:   QUEUE $\leftarrow$ NEIGHBOR.$L_m$;
 4:   for all PACKET $\in$ QUEUE do
 5:     {We will implement a drop tail system}
 6:     if PACKET.$p_d$ = STATE.$\lambda$ OR
        PACKET.$p_n$ = STATE.$\lambda$ then
 7:       if STATE.$q = q_{max}$ then
 8:         POP STATE.$L_m$;
 9:       if PACKET.$p_d \neq$ STATE.$\lambda$ then
10:         PACKET.$p_n \leftarrow$ (NEIGHBOR $\in$ NEIGHBORS
            closest to PACKET.$p_d$.);
11:         PUSH PACKET ON STATE.$L_m$;
12:       break;{If the packet was for you, do nothing; i.e., you received a packet.}
13: {Generate packets if necessary}
14: GENERATEPACKET (STATE, NEIGHBORS);
```

Algorithm 2 GENERATEPACKET: Packet generation algorithm

```
Input: Cell State STATE, Neighbors NEIGHBORS;
Output: None;
 1: if STATE.$\gamma$ = true then
 2:   {We will implement a drop tail system again}
 3:   if STATE.$q = q_{max}$ then
 4:     POP STATE.$L_m$;
 5:   PACKET $\leftarrow$; {Create a packet.}
 6:   PACKET.$p_s = \lambda$;
 7:   PACKET.$p_d = \lambda s$;
 8:   HOP_DEST $\leftarrow$ (NEIGHBOR $\in$ NEIGHBORS
      closest to server.);
 9:   PACKET.$p_n$ = HOP_DEST;
10:   PACKET.$p_t$ = DATA;{UDP does not have ACK.}
11:   PUSH PACKET ON STATE.$L_m$;
```

Algorithm 3 TCPUPDATE: TCP Update Rule

```
Input: Cell State STATE, Neighbors NEIGHBORS;
Output: None;
1: {Check neighbors for new mail packets}
2: for all NEIGHBOR $\in$ NEIGHBORS do
3:   QUEUE $\leftarrow$ NEIGHBOR.$L_m$;
4:   ACK_QUEUE $\leftarrow$ NEIGHBOR.$L_m^{ACK}$;
5:   for all PACKET $\in$ QUEUE do
6:     {We will implement a drop tail system}
7:     if PACKET.$p_d$ = STATE.$\lambda$ OR
       PACKET.$p_n$ = STATE.$\lambda$ then
8:       if STATE.$q = q_{max}$ then
9:         POP STATE.$L_m$;
10:      if PACKET.$p_d \neq$ STATE.$\lambda$ then
11:        PACKET.$p_n \leftarrow$ (NEIGHBOR $\in$ NEIGHBORS
           closest to PACKET.$p_d$.);
12:        PUSH PACKET ON STATE.$L_m$;
13:      break;
14:  for all PACKET $\in$ ACK_QUEUE do
15:    {We will implement a drop tail system}
16:    if PACKET.$p_d$ = STATE.$\lambda$ OR
       PACKET.$p_n$ = STATE.$\lambda$ then
17:      if STATE.$q = q_{max}$ then
18:        POP STATE.$L_m^{ACK}$;
19:      if PACKET.$p_d \neq$ STATE.$\lambda$ then
20:        PACKET.$p_n \leftarrow$ (NEIGHBOR $\in$ NEIGHBORS
           closest to PACKET.$p_d$.);
21:        PUSH PACKET ON STATE.$L_m^{ACK}$;
22:      else
23:        STATE.$w_{ACK} \leftarrow$ STATE.$w_{ACK} - 1$;
24:        GENERATEPACKET(STATE, NEIGHBORS);
25:        STATE.$w_{ACK} \leftarrow$ STATE.$w_{ACK} + 1$;
26:        if STATE.$w_{ACK} < 2$ then
27:          GENERATEPACKET(STATE, NEIGHBORS);
28:          STATE.$w_{ACK} \leftarrow$ STATE.$w_{ACK} + 1$;
29: {Generate packets if necessary}
30: if STATE.$w_{ACK} = 0$ then
31:   GENERATEPACKET(STATE, NEIGHBORS);
32:   STATE.$w_{ACK} \leftarrow$ STATE.$w_{ACK} + 1$;
```



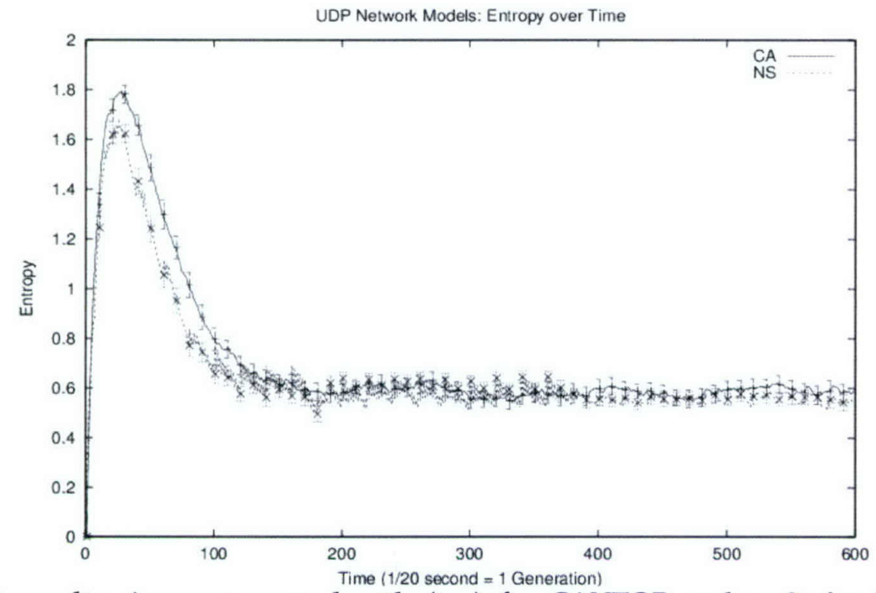
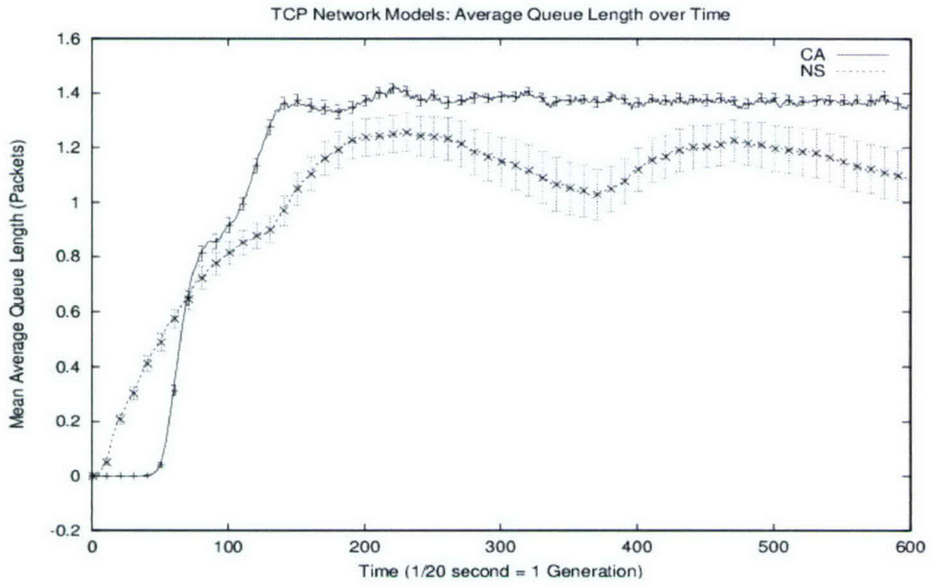


Figure 9. UDP test results. Average queue length (top) for CANTOR and ns-2 simulations trend almost identically. Amount of entropy in the network (bottom) is also very similar for both simulations



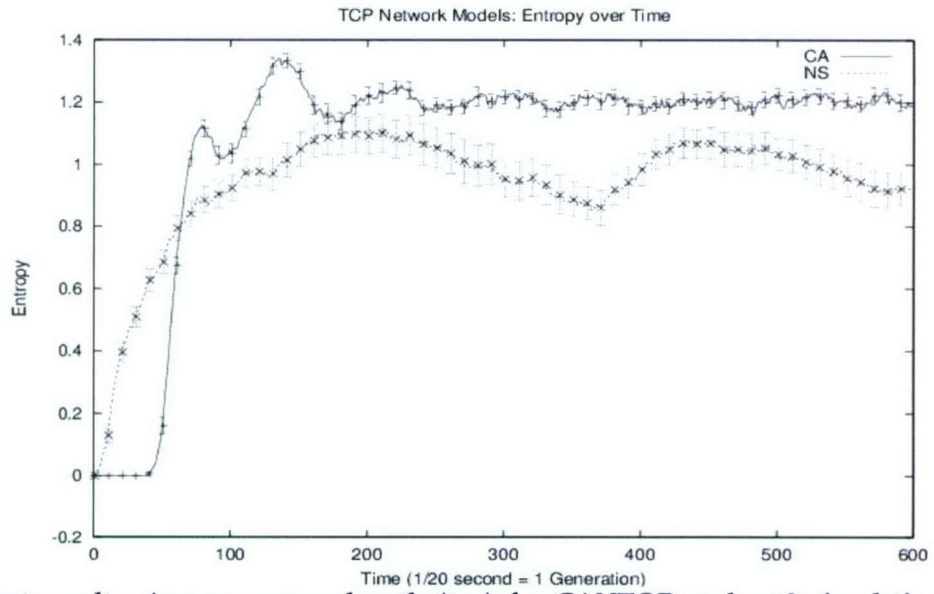


Figure 10. TCP test results. Average queue length (top) for CANTOR and ns-2 simulations trend similarly. Amount of entropy in the network (bottom) also trend similarly for both simulations

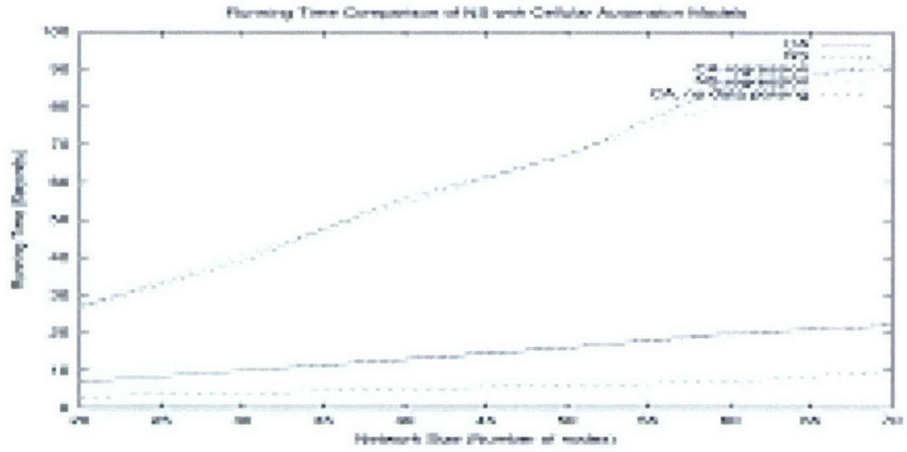


Figure 11. Run time required for ns-2 and CANTOR simulations. The top lines show the amount of time required to run ns-2 simulations as network size increases. The bottom lines show the time required for CANTOR simulations for networks of similar size.

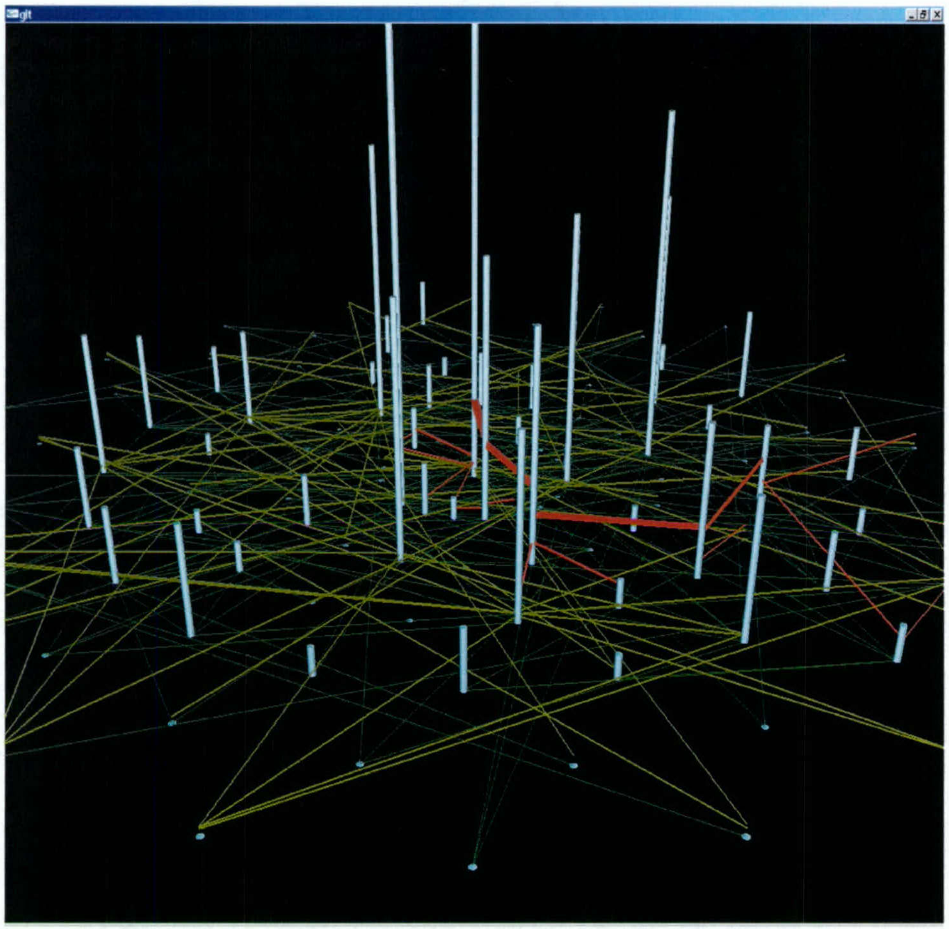


Figure 12. Example output from the improved visualization interface for CANTOR. The system can interactively display network interactions for very complex topologies in real-time.

3.4 MOBILE CODE TAXONOMY AND SECURITY ISSUES

In [Orr 2002] the taxonomy shown in Fig. 13 was developed to characterize mobile code paradigms. Each paradigm places constraints on the behavior of its systems. In the taxonomy, a transmission is a set of messages sent between threads on hosts. A system's behavior is defined as the itineraries followed by its transmissions.

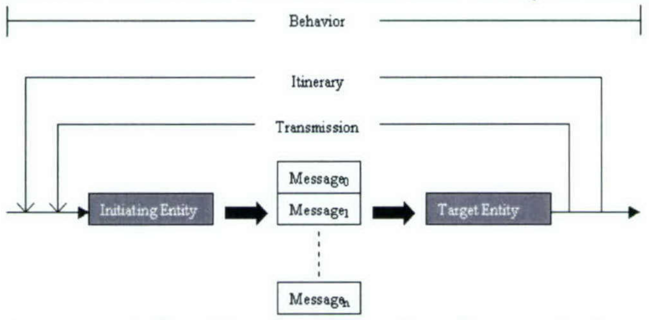


Figure 13. In the taxonomy, behavior is defined by the itineraries of transmissions. A transmission is a sequence of messages sent between threads on machines.

Fig. 14 shows the definition of a message. Each message has an instruction that signifies some action to be taken. It also has a payload signifying the (possibly empty) target of the action. In this model, resources, threads, and programs can be either fixed or mobile.

We now see that the paradigms and mobile code implementations are all limited instances of this taxonomy. For example, code on demand is limited to code requests moving from the initiating host to the target, followed by a code migrate message in return. Another example is mobile agents, which are a series of code (and state) migration requests where the agent determines the itinerary. We have used this taxonomy as the basis of an API for a flexible mobile code execution environment.

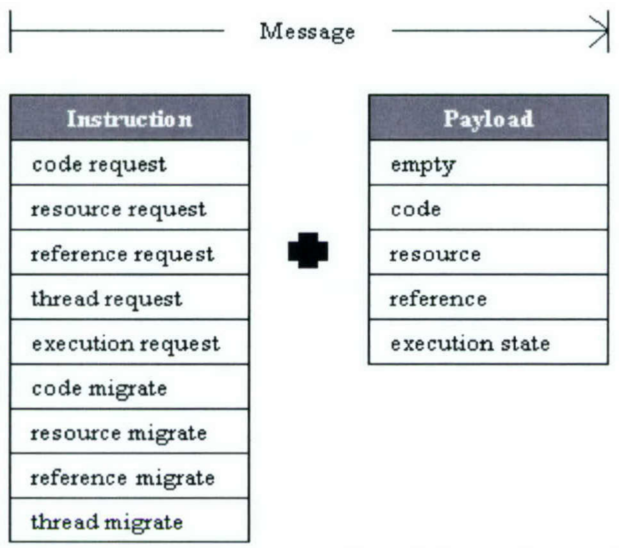
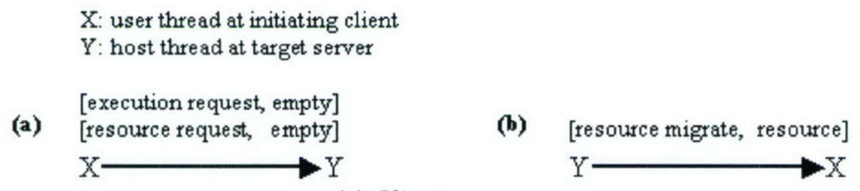
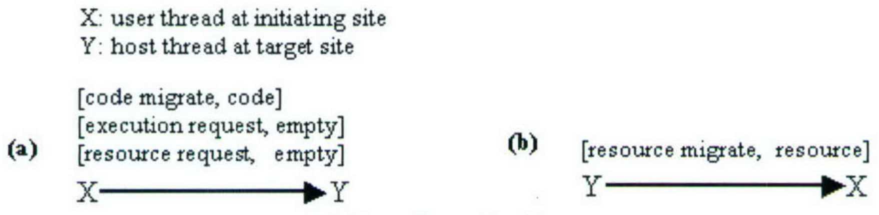


Fig. 14 Each individual message sent between threads has an instruction and a payload.

We now use the taxonomy in [Orr 2002] to group common paradigms into two families. Remember, a message contains an instruction and payload. A single transmission may contain multiple messages. Fig. 15 shows the client-server family of paradigms. In the client-server model (Fig. 15a), the client thread (X) transmits two concatenated messages to the remote thread (Y). One requests the program resource, providing data if needed. The second requests program execution. After execution, Y transmits execution results to X.


X: user thread at initiating client
Y: host thread at target server
(a)
[execution request, empty]
[resource request, empty]
X
Y
(b)
[resource migrate, resource]
Y
X
(a) Client server


X: user thread at initiating site
Y: host thread at target site
(a)
[code migrate, code]
[execution request, empty]
[resource request, empty]
X
Y
(b)
[resource migrate, resource]
Y
X
(b) Remote evaluation

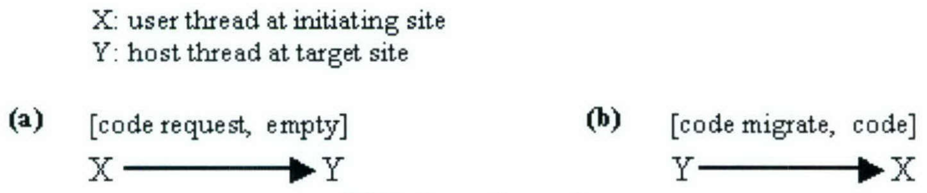
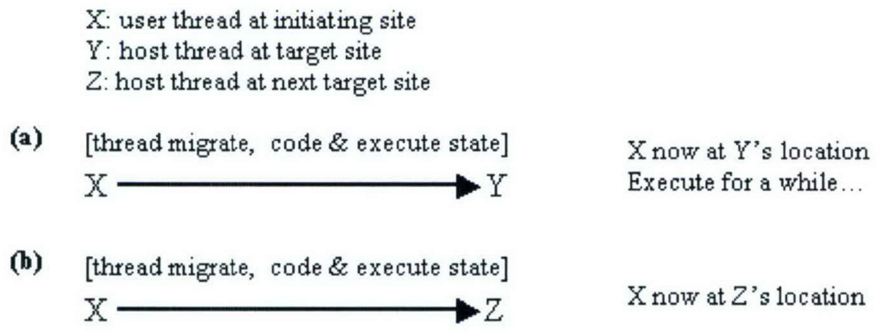

X: user thread at initiating site
Y: host thread at target site
(a)
[code request, empty]
X
Y
(b)
[code migrate, code]
Y
X
(c) Code on demand

Fig. 15. The natural progression from client-server through remote evaluation to the Java code-on-demand.

Remote evaluation (Fig. 15b) is used by CORBA factories and SOAP. Local thread (X) transmits three concatenated messages to remote thread (Y). A message containing the executable code is concatenated to a client-server style transmission. After execution, Y sends possibly NULL execution results to X.

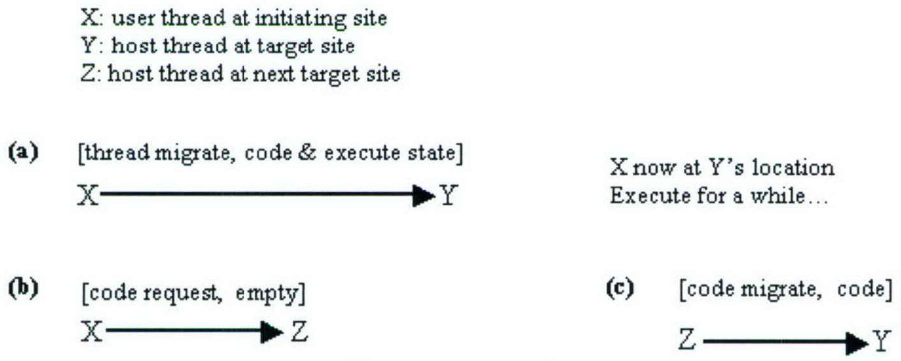
Java Applets use the code-on-demand paradigm (Fig. 15c). Local thread X transmits a single message to Y, requesting a code download. Thread Y transmits a message to X that contains the code. X executes the code locally.

In contrast to the client server family, characterized by users initiating action and a reactive infrastructure, the agent family supports autonomy and adaptation within the infrastructure. Fig. 16 shows the agent mobile code paradigm family.



X: user thread at initiating site
Y: host thread at target site
Z: host thread at next target site
(a)
[thread migrate, code & execute state]
X
Y
X now at Y's location
Execute for a while...
(b)
[thread migrate, code & execute state]
X
Z
X now at Z's location

(a) Mobile agent



X: user thread at initiating site
Y: host thread at target site
Z: host thread at next target site
(a)
[thread migrate, code & execute state]
X
Y
X now at Y's location
Execute for a while...
(b)
[code request, empty]
X
Z
(c)
[code migrate, code]
Z
Y

(b) Active network

Fig. 16. Mobile agents, process migration, and active networks are another family of distributed systems, where more autonomy is given to the distributed system.

The mobile agent paradigm (Fig. 16a) uses two threads for each hop. Thread X executes locally and composes a thread migrate message containing agent code and state. This message is transmitted to thread Y on the remote host, where execution continues. A set of n hops requires n transmissions between up to n+1 threads. The agent decides when and where to migrate. The process migration paradigm differs from the agent paradigm in one way. The local host decides when and where the process migrates instead of the agent.

Active networks include many paradigms. In one, packets executed while traversing the network. This is a type of process migration. In another paradigm, packets reprogrammed network infrastructure. This (Fig. 16b) combines mobile agent and code on demand paradigms.

The main approaches currently used to maintain mobile code security are [Rubin 1998]:

- *Sandbox* – limit the instructions available for use.
- *Code signing* – ensures that code is from a trusted source.
- *Firewalls* – limits the machines that can access the Internet.
- *Proof Carrying Code (PCC)* – code carries an explicit proof of its safety.

The first three approaches are in widespread use. Netscape and Sun browsers use a hybrid approach that combines use of a sandbox and code signing. Firewalls are in widespread use, but have serious limitations on their ability to detect malicious code. It is not clear that generic implementations of PCC will ever be possible.

The approaches listed above look solely at protecting hosts from malicious code. Little has been done to protect code from malicious hosts. Methods recorded in the literature include:

- *Computing with encrypted functions* - It has been shown that it is possible in some cases to execute encrypted functions on encrypted data.
- *Code obfuscation* - With obfuscation, the object code is deliberately scrambled in a way that keeps it functional but hard to reverse engineer.
- *Itineraries* - Itineraries can be kept of the nodes visited by a mobile code package.
- *Redundancy* - Multiple code packages can work in parallel on multiple hosts and compare their results.

- *Audit trail* - Partial results can be logged throughout a distributed computation.
- *Tamper-proof hardware* – viruses or other methods can not corrupt hosts that are tamper proof.

It is worth noting that widespread network attacks tend to involve some type of mobile code. Viruses and worms are a danger almost entirely due to their ability to migrate from host to host. The fact that we are still confronted by viruses and worms illustrates the widespread security measures are not working. They may be inadequate, or just poorly implemented.

Our mobile code taxonomy was based on a security incident taxonomy developed at Sandia National Laboratories [Howard 1998]. It was developed as a language for describing security incidents. Using the terminology of this language, security intrusion descriptions are unambiguous. The taxonomy is shown in Fig. 17

We describe the taxonomy and map it to mobile code using our taxonomy. Each security incident is a combination of one or more attacks; perpetrated by a group to fulfill its objectives. Attacks use tools to exploit system vulnerabilities and create an unauthorized result. Each unauthorized result is produced by an event. Events are the actions an attacker takes to exploit the vulnerabilities of specific targets. Fig. 17 enumerates the most common possibilities for every element of the taxonomy.

The behavior of a malicious mobile code package results in a single security incident. The itinerary of the package behavior is a set of transmissions. Each transmission used by the malicious code is an attack, and every message is a security event. Each instruction is an action applied to a payload, which is a potential target. Unauthorized mobile code executions produce unauthorized results.

Where do mobile code security measures fit in? A sandbox contains code execution. It protects a target machine from unauthorized access. A firewall's goal is to protect a target sub-network from unauthorized access. Proof carrying code's goal is to allow target machines to reject offensive code before executing the code.

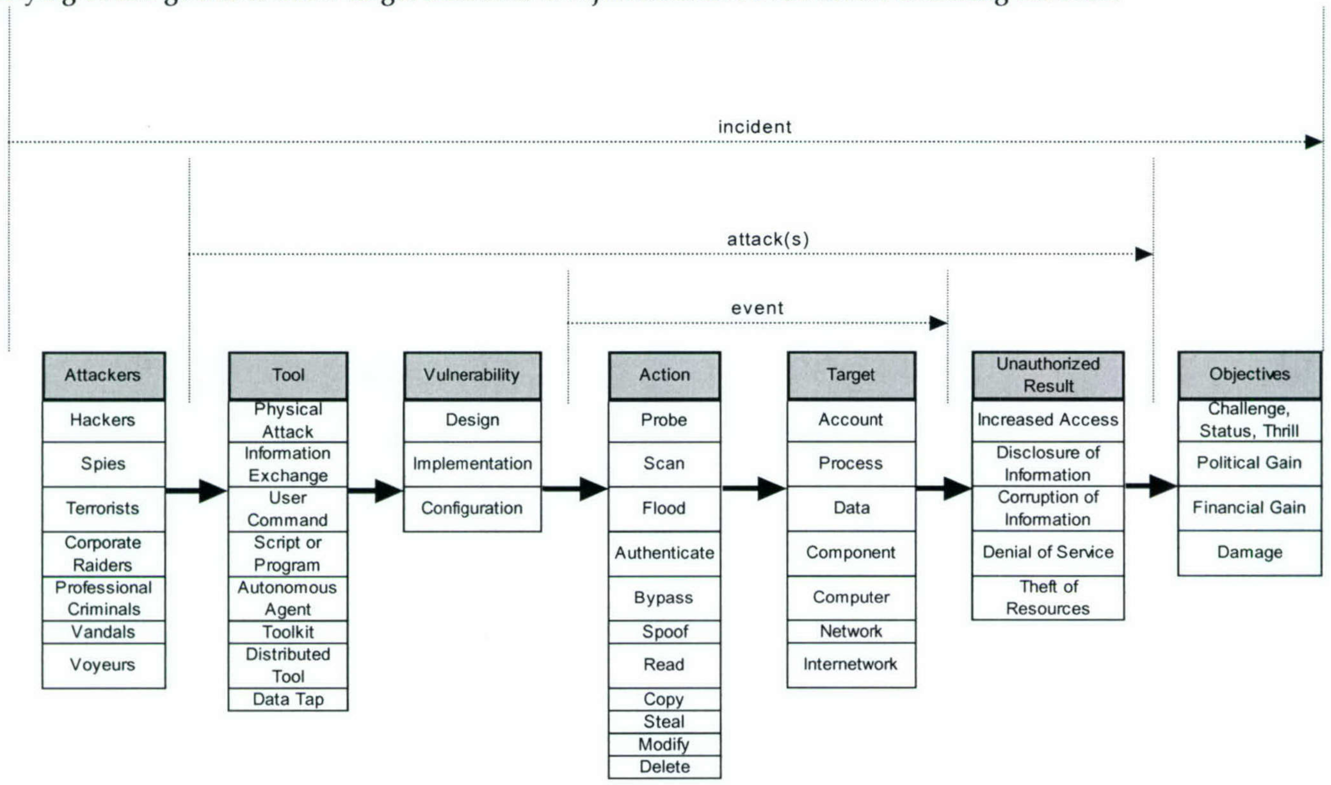


Figure 17. Security taxonomy shows that attackers use tools to exploit vulnerabilities. Actions are then taken against targets to produce unauthorized results fulfilling the attacker's objectives. Note how events in this taxonomy correspond to messages in the taxonomy from [Orr 2002].

Although a case could be made that these approaches remove vulnerabilities, in essence all these approaches protect target machines, or networks, from attacks.

Code signing works at a different level. By identifying the source of a program, code may be rejected as being unsafe. Alternatively if code is found to be malicious, the signature can be a forensics tool for proving culpability

Some approaches for protecting code from hosts in section IV similarly concentrate on fortifying components. Computing with encrypted functions and code obfuscation protect mobile code programs from being targets by making them difficult to decipher.

Tamper-proof hardware makes system corruption impossible, removing an entire class of vulnerabilities. This allows both host and code to trust the tamper-proof component. In the ideal case, this protects both from being targets of attack.

The use of itineraries, redundancy, and audit trails work at an entirely different level. Although each single event in a mobile code intrusion is of relatively minor importance, the consequences of the aggregate behavior can easily become catastrophic. These approaches look at aggregates of messages, and thus work closer to the incident or behavior levels of the taxonomies.

Comparing taxonomies of mobile code and security incidents shows the relationship of security measures and system vulnerabilities to a given mobile code approach. Most security measures fortify potential targets of attacks. While this is important and necessary, consider the larger picture. Many e-mail viruses do not perform actions that are forbidden by a sandbox. Worms primarily exploit software implementation errors. It is unlikely that software design will advance in the near future, if ever, to the point where we automatically foresee the abuses of software features or consistently produce bug-free systems.

4. DISTRIBUTED SYSTEMS DESIGN AND IMPLEMENTATION

This section discusses the work done on this project related to the design of an adaptive, survivable infrastructure. This infrastructure is designed to use the positive aspects of mobile code and peer-to-peer (P2P) networks. First we illustrate how to estimate the Quality of Service (QoS) of P2P systems. We then discuss how to calculate phase transitions in systems of this type, where local behaviors aggregate to define global system behaviors. We then delve into implementation details of our prototype dynamic battle management system.

4.1 P2P QoS

We are interested in creating a network infrastructure capable of adapting to malicious, possibly catastrophic events. Mobile code technology enables transmission and execution of programs between networked nodes. It supports adaptation by allowing nodes to reconfigure their software and change roles dynamically. P2P networks distinguish themselves from traditional client/server or master/slave networks in that there is neither a central point of control nor centralization of data. They potentially support adaptation by allowing network structure to evolve.

As described in [Oram 2001], many technologies can be classified as P2P. The two most widely known implementations are Napster and Gnutella. Napster is a file-sharing network with only one central index. This index contains a database of users and their files. When a user connects to Napster, a list of files available on the user's machine is added to a central index. When the user requests a specific file, a list of participating machines containing the file is returned. The file can be retrieved from any machine on the list. This is an efficient architecture. File names and machine addresses contain tens of bytes. Files being exchanged typically contain megabytes of data. The large data transfers occur between machines chosen virtually at random. This tends to spread data traffic evenly throughout the Internet. On the other hand, its survivability is poor as a single failure or a court order can stop the entire network by switching off the central index. Gnutella offers a radically different approach. It is fully distributed with no single point of failure. Each node has an index of its own files. File discovery is performed by flooding the network with request packets. There appear to be serious scalability issues with this approach. On the other hand, Gnutella has interesting survivability characteristics. To stop the Gnutella service, it would be necessary to stop every node on the Internet running Gnutella.

On the same lines as Napster and Gnutella, we consider a P2P network where the nodes represent computers, work stations, or servers storing mobile code (analogous to audio files in Napster) and the arcs represent physical or logical connectivity between two nodes. However, one of the main differences is that we use a protocol with a "time-out" parameter for abandoning a search. We studied the question: what is an appropriate number of indexes for a P2P network? We analyze this problem in terms of performance, scalability, and survivability. Recall that Napster has 1 index that stores the locations of all its files. It is efficient, but has a single point of failure. Gnutella provides n indexes for n nodes. It lacks single points of failure, but does not scale well. We consider Napster and Gnutella as the extreme cases of a continuum. The number of indexes varies in the range of 1 to n. Another question studied in this paper is, what is the appropriate time-out value to use?

Before designing and implementing a network, it is necessary to thoroughly analyze its performance and dependability. These two issues are critical in P2P networks because of their complex topologies and the chaotic environments in which they operate. In P2P networks, it is of interest to know how long it will take to locate and retrieve a file (in our case mobile code) and what percentages of the requests are lost. These issues become more critical with each additional hop a request needs to travel, since nodes may become unavailable due to random failures or attacks.

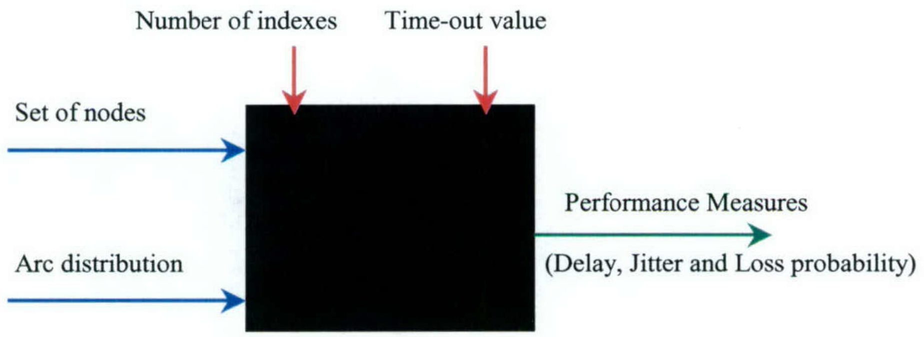


Figure 18. Problem description

Figure 18 illustrates the problem we studied. Parameters on the left side of the black box are inputs over which we have no control. The set of nodes and arc distribution are fixed prior to determining system parameters. Although we do not explicitly state link and computing (i.e. node) speeds as inputs, we assume their values are known. Inputs at the top of the box are controllable variables; we choose the number of indexes and time-out values to optimize performance. The arc exiting the box represents the objective function (weighted sum of QoS and dependability measures) to be optimized. We assume nodes fail randomly and independently. Nodes fail for several reasons including denial-of-service attacks.

Our results only require the node set and arc distribution as input to the models but not the type of network. However, in order to generate examples of networks for our numerical results, we resort to three classes of randomly generated graphs based on the literature. It is important to note that network types are merely to classify our results, but our analysis can handle any network structure, not just the following:

- ***Random graphs or Erdös-Rényi graphs*** – A class of graphs with probability $\hat{p}$ an arc exists between any two nodes. Node degree distribution follows Poisson distribution as $n \to \infty$. The average number of hops between nodes grows proportionately to the logarithm of the number of nodes.
- ***Small world graphs*** – A class of graphs with two properties: (*i*) average number of hops increases with the number of nodes in the same order of magnitude as random graphs, and (*ii*) there is a significant clustering of nodes (i.e. many nodes have multiple neighbors in common). For this class, we use the connected caveman model described in [Watts 1999]. A set of fully connected components is constructed. One arc at random is rewired in each fully connected component so that the set of components is connected in a cycle. A small set of arcs in the resulting structure is rewired at random. The node degree distribution depends on the number of arcs re-wired.
- ***Scale-free graphs*** – A class of graphs where the probability $p(k)$ that a node has degree k follows a power law distribution $p(k) \propto k^{-\gamma}$, where γ is a constant. Empirical studies have shown that many real-world systems including the Internet and Gnutella have this property. The average number of hops of scale-free networks grows more slowly with respect to the number of nodes than for Erdös-Rényi graphs.

These three models are general and applicable to many applications. They all scale well. The number of hops increases at most logarithmically with respect to the number of nodes. The existence of nodes with very large degrees in scale-free networks makes these networks vulnerable to disruptions when those nodes fail. Erdös-Rényi and small-world graphs rarely have nodes with large degrees and may be more suitable for survivable systems. Also, the clustered nature of small world graphs could be useful for detecting and containing system intrusions.

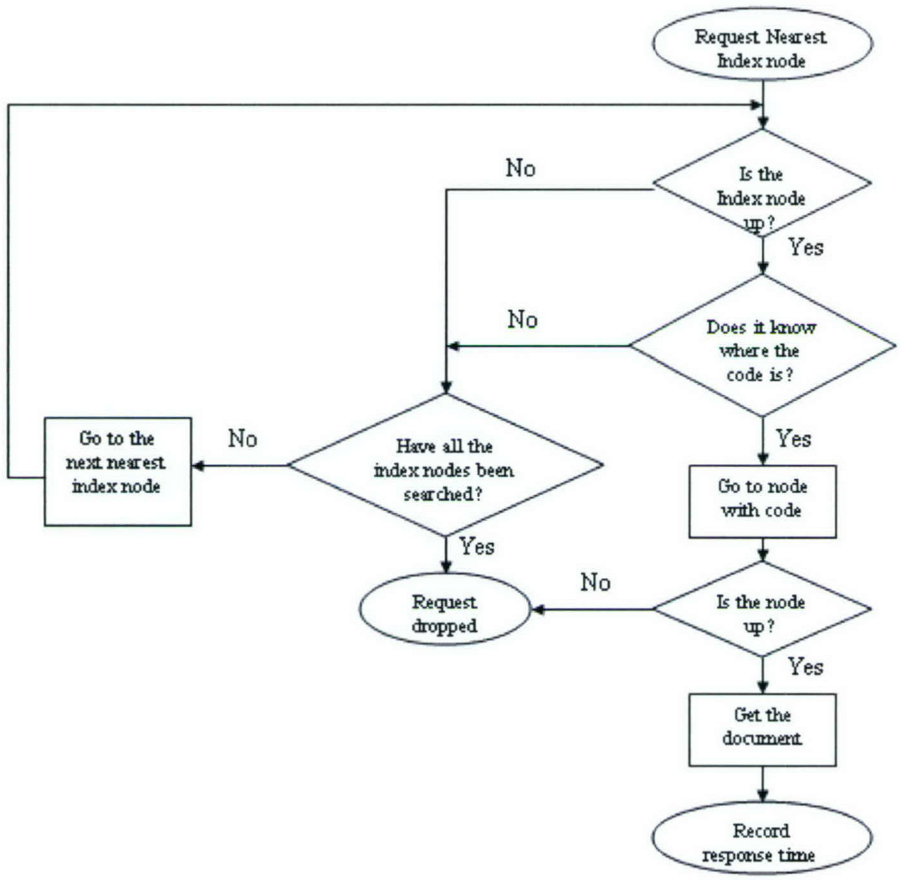

Request Nearest Index node
Is the Index node up?
No
Yes
Does it know where the code is?
No
Yes
Have all the index nodes been searched?
No
Go to the next nearest index node
Yes
Go to node with code
Is the node up?
No
Request dropped
Yes
Get the document
Record response time

Figure 19. Flowchart of request-response process

We divide the network into approximately equal sized groups per index. Each index node has a database of the files available from nodes in its group. The source node requests a particular document from its index node and starts a timer. (An index node can also be a source node, in which case the travel time to the first index node is zero.) If the index node knows the location of the requested document (i.e. the document and source node are in the same group), it informs the source node of the location of the node containing the document (*destination node*). Otherwise, the index node sends the location of the next nearest index node to the source node. The source node queries index nodes for the document until all indexes have been searched. If the destination node is identified, the source node requests the document from it. If a transfer is completed before the timer expires then it's a success, otherwise the request is lost. A request is also lost if an index node with document information is down or the destination node is down. In case of a loss, the source node does not retry the same request. The request-response process is shown in Figure 19.

Before deriving performance measures it is necessary to estimate the expected number (and variance) of hops between nodes in the network. Our approach for doing this is derived in [Kapur 2002]. The expected number of hops between any two nodes chosen at random can be estimated as:

$$\mu_{hops} = \left(\frac{q_1 + 2(q_2 - q_1) + 3(q_3 - q_2) + \ldots}{n} \right)$$

$$\mu_{hops} = \frac{1}{n}\left(k_{ave} + 2\left((1-C)M[2]\left(\sum_{i=2}^{n-1} p_i i(i-1) \right) - k_{ave} \right) + \right.$$

$$\left. \sum_{h=3}^{Maxhops} h\left(\left((1-C)^{h-1} \left(\prod_{i=2}^{h} M[i] \right) \sum_{i=2}^{n-1} p_i i(i-1)^{h-1} \right) - \left((1-C)^{h-2} \left(\prod_{i=2}^{h-1} M[i] \right) \sum_{i=2}^{n-1} p_i i(i-1)^{h-2} \right) \right) \right)$$

Given the mean (μ_{hops}) and the distribution of the hop count, the standard deviation (σ_{hops}) can be computed using the following formula:

$$\sigma_{hops} = \sqrt{\frac{\sum_{i=1}^{n} \sum_{j=1}^{n} (h_{ij} - \mu_{hops})^2}{n^2 - 1}}$$

where, h_{ij} is the number of hops from node i to node j and n is the number of nodes in the network.

Table 1 below compares the analytical and simulation results (in Matlab) for the mean and standard deviation of number of hops for the 3 networks: Erdös-Rényi graph, small world and scale-free. The number of nodes and average arc degree for the 3 networks were: Erdös-Rényi graph, 25 nodes and 3 average arc degree, small world, 25 nodes and 4.4 average arc degree and scale-free, 25 nodes and 5.28 average arc degree. 10 different networks were created for each of the 3 types with the same above-mentioned topology. In Matlab the hop distribution was computed using Dijkstra's Shortest Path Algorithm [23]. Table 1 shows that the analytical and simulation results for mean and standard deviation of hops are quite close, hence validating our analytical model.

	Simulation		**Analytical**	
	Mean	*Std. Dev.*	*Mean*	*Std. Dev.*
Erdös-Rényi Graph	2.7307	1.2985	2.7478	1.3409
Small World	2.5543	1.2044	2.58	1.296
Scale-free	1.9091	0.7518	1.904	0.747

Table 1: Comparison of *mean* and *standard deviation* for the three networks

We now use the mean and the variance of the number of hops derived in Section 3 to obtain performance measures such as *delay, jitter* and *loss probability*. We then use them to compute the optimal number of indexes and time-out value. We consider several scenarios. In particular, we derive *delay, jitter* and *loss probability* in for a scenario without queuing at the nodes and an infinite time-out value. We then incorporate queues at index nodes and destination nodes with no node failures. No requests are lost. At which point, we incorporate time-outs. We compare the analytical and simulation models for the scenario with both infinite and finite time-outs.

Since the number of indexes (I) and the time-out values are design variables, we would like to select them optimally. To do so, we first build analytical models relating performance measures to the design parameters. We initially assume an infinite time-out and infrequent request arrival so there is no resource contention (i.e. queuing) among requests.

Let T be the time to receive a response to a request. The expected delay is given by:

$$E(T) = \left(\frac{1}{1-(1-p)^{co}} \right)\left(\frac{2*q}{I} * \sum_{i=1}^{I} \mu_i(n, k_{ave}, s) * p_i + \frac{p*d*mo*co}{I} \sum_{i=1}^{I} p_i(i-1) \right.$$

$$+0.5*d*mo*co(\frac{1}{n}+\frac{1}{I})\sum_{i=1}^{I} p_i + (\frac{q}{l}+\frac{E(B)}{l})\sum_{i=1}^{I}\alpha_i(n,k_{ave},s)p_i\Big)$$

where,

p_i = *P (Code location is found in index i) For i = 1, 2, …I*

$$= \sum_{r=1}^{i} \frac{{}^{(i-1)}C_{(r-1)} * {}^{(I-co)}C_{(r-1)}}{{}^{(I)}C_{(r-1)}} * p^r * (1-p)^{i-r} * min\left[\frac{co}{I-r+1},1\right]$$

(with the understanding that ${}^iC_r = 0$ if $i < r$) and

p_0 = *Probability that all the indexes with the code are down* = $1-\sum_{j=1}^{I} p_j = 1-(1-p)^{co}$

The jitter of the response time is given by

$$Jitter = \sqrt{Var(T)}$$

To compute *Var(T)*, we use the relation $Var(T) = E(T^2) - [E(T)]^2$ and $E(T)$ is from equation (19). In order to obtain $E[T^2]$, we define T_i as the random variable denoting the time to retrieve the document conditioned upon the document indexed in index *i*, therefore we have the following: the expressions for $E(T^2)$, $Var(T_i)$ and $E(T_i)$ are:

$$E(T^2) = \frac{1}{(1-(1-p)^{co})}\sum_{i=1}^{I}\{Var(T_i)+[E(T_i)]^2\}p_i,$$

$$Var(T_i) = \frac{4*\sigma_i^2(n,k_{ave},s)*q^2}{l^2} + \left(\frac{d*mo*co}{I}\right)^2 (i\text{-}1)*p*(1\text{-}p) + \frac{1}{12}*\left(\frac{d*mo*co}{I}\right)^2$$

$$+\frac{\beta_i^2(n,k_{ave},s)*q^2}{l^2} + \left(\frac{0.5*d*mo*co}{n}\right)^2 + \frac{1}{l^2}\{Var(B)\ \beta_i^2(n,k_{ave},s)$$

$$+ Var(B)*\alpha_i^2(n,k_{ave},s) + E(B)^2*\beta_i^2(n,k_{ave},s)\}, \text{ and}$$

$$E(T_i) = \left(\frac{2*q*\mu_i(n,k_{ave},s)}{l} + p(i-1)\frac{d*mo*co}{I} + \frac{0.5*d*mo*co}{I} + \frac{\alpha_i(n,k_{ave},s)*q}{l}\right.$$

$$\left.+\frac{0.5*d*mo*co}{n} + \frac{\alpha_i(n,k_{ave},s)*E(B)}{l}\right).$$

Since the time-out is infinite at this point of the analysis, requests will be lost only if the index node with mobile code information or the destination node is down. The proportion of requests lost or loss probability (L_S) will be:

$$L_s = 1 - p_d[1-(1-p)^{co}]$$

where, $1-(1-p)^{co}$ is the probability of the index node with code information being down, and p_d is probability of destination node being up, a function of the number of index nodes.

So far we have considered a scenario where the nodes go up and down from time to time and requests arrive very infrequently so that there is no contention among requests for resources, this is especially the case at the index nodes and the destination nodes that could potentially receive a large number of requests. Now we consider the case where no nodes fail and there is resource contention so requests may have to be stored in a queue at the indexes and destination nodes. Queues at index nodes and destination nodes are approximated as M/G/1 queues with Poisson inter-arrivals, uniformly distributed service times and a single server. To ensure the stability of the M/G/1 system, a lower limit on the inter-arrival times was determined given the average service time at index nodes. The expected delay with queues at the index nodes and destination nodes is given by:

$$E(T) = \left(\frac{2q}{l} \sum_{i=1}^{I} \mu_i(n, k_{ave}, s) + pE(W_I) + E(W_I) + E(W_n) + \left(\frac{q}{l} + \frac{E(B)}{l}\right) \sum_{i=1}^{I} \alpha_i(n, k_{ave}, s) \right)$$

where, $E(W_I)$ is the average waiting time a randomly arriving request spends in index node queue plus the expected time to search the index node, and $E(W_n)$ is the average waiting time a randomly arriving request spends in destination node queue plus the expected time to search the destination node.

The jitter with queues at index nodes and destination nodes is given by: $Jitter = \sqrt{Var(T)}$

where, $Var(T) = E(T^2) - [E(T)]^2$,

$E(T)$ is from equation (22), $E(T^2) = Var(T_i) - [E(T)]^2$, with

$$Var(T_i) = \frac{4 * \sigma_i^2(n, k_{ave}, s) * q^2}{l^2} + \left(\frac{d * mo * co}{I}\right)^2 + \frac{1}{12} * \left(\frac{d * mo * co}{I}\right)^2 + E(I) * Var(W_q)$$

$$+ \frac{\beta_i^2(n, k_{ave}, s) * q^2}{l^2} + \left(\frac{0.5 * d * mo * co}{n}\right)^2$$

$$+ \frac{1}{l^2}\{Var(B)\ \beta_i^2(n, k_{ave}, s) + Var(B) * \alpha_i^2(n, k_{ave}, s) + E(B)^{2} * \beta_i^2(n, k_{ave}, s)\}, \text{ and}$$

$$E(T_i) = \frac{2q\mu_i(n, k_{ave}, s)}{l} + E(W_I) + 0.5E(W_I) + \frac{\alpha_i(n, k_{ave}, s)q}{l} + 0.5E(W_n) + \frac{\alpha_i(n, k_{ave}, s) * E(B)}{l}$$

If the time-out value was finite, then requests would also be lost if the time to get a response exceeds the time-out value. Since the delay is a sum of a large number of independent random variables, we can approximate using the central limit theorem that $T \sim$ Normal [E(T), Var(T)]. Now the probability that time-out occurs even when the document is available will be

$P(T > \theta) = 1 - P(T \leq \theta)$

where, θ is the finite time-out value. The above expression can be written as

$$P(T > \theta) = \varepsilon = 1 - \Phi\left[\frac{\theta - E(T)}{\sqrt{Var(T)}}\right]$$

where Φ can be obtained from z tables for Normal Distribution. Then, the response time given that document is retrieved before time-out ($E(T_\theta)$) will be

$$E(T_\theta) = \frac{\int_{-\infty}^{\theta} x f(x) dx}{(1-\varepsilon)}$$

where f(x) is the normal probability density function. The variance will be

$$Var(T_\theta) = \left\{ \frac{(1-\varepsilon) \int_{-\infty}^{\theta} x^2 f(x)\, dx - \varepsilon (E(T_\theta))^2}{(1-\varepsilon)^2} \right\}$$

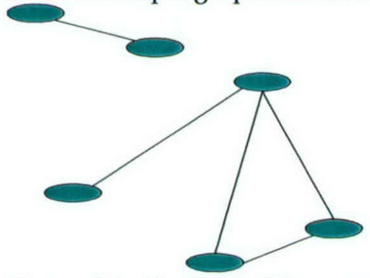
The loss probability will be $L_\theta = 1 - p_d[1-(1-p)^{co}](1-\varepsilon)$

Comparison of these analytical results with simulations can be found in [Kapur 2002].

4.2 Analysis of Random Network Designs

A graph is traditionally defined as the tuple [V,E]. V is a set of vertices, and E is a set of edges. Each edge *e* is defined as *(i,j)* where *i* and *j* designate the two vertices connected by *e*. In this paper, we consider only undirected graphs where *(i,j)*=*(j,i)*. (Many systems are modeled using directed graphs (di-graphs) where *(i,j)* ≠ *(j,i)*.) An edge *(i,j)* is incident on the vertices *i* and *j*. We do not consider multi-graphs where multiple edges can connect the same end-points. We use the terms vertex and node interchangeably. Edge and link are also used synonymously.

Many data structures have been used as practical representations of graphs. Common representations and their uses can be found in [11]. For example, a graph where each node has at least one incident edge can be fully represented by the list of edges. Another common representation of a graph, which we will explore in more depth, is the connectivity matrix. The connectivity matrix *M* is a square matrix where each element *m(i,j)* is 1 (0) if there is (not) an edge connecting vertices *i* and *j*. For undirected graphs this matrix is symmetric. **Figure 20** shows a simple graph and its connectivity matrix.



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 20. On the left is a graph of six nodes. On the right is its associated connectivity matrix. Row *j* of the matrix corresponds to the *j*th node from the top.

As a matter of convention, the diagonal of the matrix can consist of either zero's or one's. One's are frequently used based on the simple assertion that each vertex is connected to itself.. We use a convention where the diagonal is filled with zeros.

A *walk* of length *z* is a set of edges, which can be expressed as an ordered list of *z* edges $((i_0,j_0),(i_1,j_1),\dots,(i_z,j_z))$, where each vertex j_a is the same as vertex i_{a+1}. A *path* of length *z* is a walk where all i_a are unique. If j_z is the same as i_0, the path forms a *cycle*.

A connected component is a set of vertices such that from any vertex in the component there is a path to all other vertices in the component. (In the case of di-graphs, this would be a fully connected component.) A complete graph has an edge directly connecting any two vertices in the graph. A complete subgraph is a subset of vertices in the graph with edges directly connecting any two members of the set.

One interesting property of connectivity matrices we use is the fact that element $m^z(i,j)$ of the power *z* of graph *G*'s connectivity matrix *M* (i.e. M^z) is the number of walks of length *z* from vertex *i* to vertex *j* that exist on *G* [Cvetovic 1979]. This can be verified using the definition of matrix multiplication and the definition of the connectivity matrix. Iterative computation of M^z until $M^z=M^{z-1}$ can be used to find the connected components in a graph.

We now show how to construct connectivity matrices for analyzing classes of random and pseudo-random graphs. The first model we discuss is the Erdös-Rényi random graph. It is provided for completeness as it is the most widely studied class, and as a tutorial since it is the simplest class. Erdös-Rényi random graphs are defined by the number of nodes n and a uniform probability p of an edge existing between any two nodes. Let's use E for $|E|$ (i.e. the number of edges in the graph). Since the degree of a node is essentially the result of multiple Bernoulli trials, the degree of an Erdös-Rényi random graph follows a Bernoulli distribution. Therefore as n approaches infinity, the degree distribution follows a Poisson distribution. Figure 21 shows different embeddings of an example Erdös-Rényi graph.

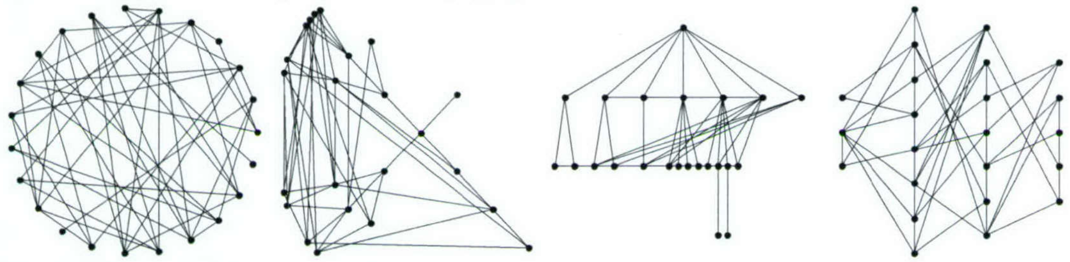


Figure 21. Example Erdös-Rényi graphs with n equal to 23 nodes and the probability p equal to 0.2. From left to right: nodes in a circle, radial embedding, rooted embedding from a random node, ranked embedding by geodesic distance from three nodes chosen at random.

It has been shown that the expected number of hops between nodes in this graph grows proportionally to the log of the number of nodes. Note that Erdös-Rényi graphs do not necessarily form a single connected component. When $E - n/2 << -n^{2/3}$ the graph is in a sub-critical phase and almost certainly not connected. A phase change occurs in the critical phase where $E= n/2 +O(n^{2/3})$ and in the supercritical phase where $E-n/2 >> -n^{2/3}$ a single giant component becomes almost certain. When $E= n \log n/2+ O_p(n)$ the graph is fully connected [15]. (Note that the expected number of edges for an Erdös-Rényi graph is $n(n-1)\, p\, /2$).

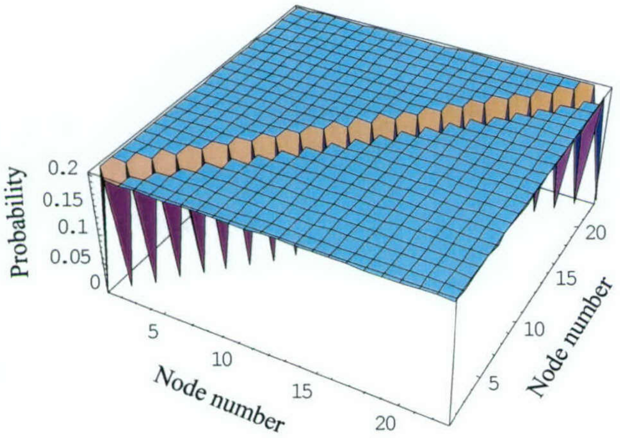


Figure 22. A three-dimensional plot of the probabilistic connectivity matrix for Erdös-Rényi graphs with n=23 and p=0.2. The diagonal values are zero. All other edges have the same probability.

To construct a probabilistic connectivity matrix for this graph, create an n-by-n matrix with all elements on the diagonal set to zero and all the other elements set to p. For example, if n is 3 and p is 0.25, we get:

$$\begin{bmatrix} 0 & 0.25 & 0.25 \\ 0.25 & 0 & 0.25 \\ 0.25 & 0.25 & 0 \end{bmatrix}$$

We now consider the scale-free model. It comes from empirical analysis of real-world systems, such as e-mail traffic, the World Wide Web, and disease propagation. See [Albert 2001] for details. It is appropriate for some mobile systems that evolve over time. In this model, the node degree distribution varies as an inverse power law (i.e. P[d] $\propto d^{-\gamma}$). These graphs are called scale-free because the power law structure implies that nodes exist with non-zero probability at all possible scales. The expected number of hops for scale-free networks is smaller than the expected number of hops for Erdös-Rényi graphs. Scale-free (SF) graphs are defined by two parameters: number of nodes n, and scaling factor γ□ (see Figure 23). Of the random graph classes considered here, node degree variance in this class is the largest.

Many existing systems are SF networks. The Internet is SF. Empirical analysis done by different research groups at different times find the Internet's γ parameter value ranging from 2.1 to 2.5). Studies of SF networks indicate their structure has unique dependability properties [4]. Epidemiological studies show parallels between biological pathogen propagation and computer virus propagation in SF graphs like the Internet.

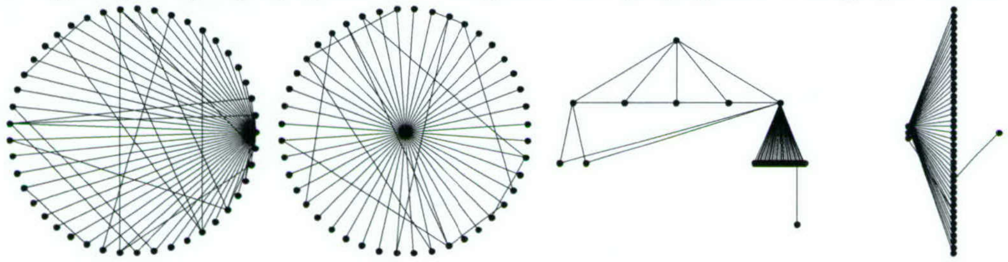


Figure 23. Example scale-free graphs with n=45 and γ=3.0. From left to right: nodes in a circle, radial embedding, rooted embedding with the root set as the second largest hub, and ranked embedding in order of the geodesic distance from the three largest hubs.

Figure 23 illustrates how scale-free graphs differ from Erdös-Rényi graphs. The majority of nodes have degree one or two, but there exists a small number of hub nodes with a very large degree. Erdös-Rényi graphs have an almost flat architecture with node degree clustered about the mean. The hub nodes dominate the topology of the scale free graphs. The ranked embedding illustrates that it is extremely unlikely that a node would be many hops away from a major hub.

An algorithm for constructing these graphs based on positive feedback that produces graphs with $\gamma \approx 3$, can be found in [Barabasi 1999]. Barabási's use of positive feedback plausibly explains how SF systems emerge and why they are widespread. We present a method for constructing SF networks that uses positive feedback to produce graphs with arbitrary scaling factors. Utilizing the mechanism believed responsible for creating SF networks has two advantages: (*i*) it may produce graphs closer to those found in reality, and (*ii*) it helps explain how SF systems work.

Creating a probabilistic connectivity matrix for scale-free graphs is more challenging. Remember that scale-free graphs are characterized by n (the number of nodes) and γ (the scaling factor). The first step is to compute a probability distribution for node degree d. Remember $P[d] \propto d^{-\gamma}$. We compute the probability distribution, by finding a constant factor so that all probabilities sum to 1. Set:

$$P[d] = bd^{-\gamma}$$

Since node degree ranges from 1 to n-1:

$$1 = \sum_{d=1}^{n-1} bd^{-\gamma}$$

thus:

$$b = 1 \Big/ \sum_{d=1}^{n-1} d^{-\gamma}$$

We now have a closed form solution for the node degree probability distribution. The next step is determining how many edges are incident on each node. First construct a vector v of n-1 elements, whose values range from 0 to 1. Each element k of the vector contains the value:

$$v[k] = \sum_{d=1}^{k-1} bd^{-\gamma}$$

Vector element v[0] has the value zero and element v[n-1] has the value one. Each element v[k] represents the probability of there being a node of degree less than or equal to k. Each row of the probabilistic connectivity matrix represents the expected behavior of $1/n^{th}$ of the nodes of the class under consideration. We now construct a vector v' of n elements, the value of $v'[k]$ states how many edges are incident on node k. Set $v'[k]$ to the index of the largest element of v whose value is less than or equal to k/n

The elements of the connectivity matrix are probabilities of connections between individual nodes. These values are computed using the insight from [16] that scale-free networks result from positive feedback. Nodes are more likely to connect to other nodes with many connections. The value of each matrix element (k, i) is therefore:

$$P[k,i] = \frac{v'[i]v'[k]}{\sum_{m \neq k} v'[m]}$$

The likelihood of choosing another node i to receive a given edge from the current node k is the degree of i divided by the sum of the degrees of all nodes except k. Summing these factors would give a total probability of one for the row. Since k has degree $v'[k]$ these probabilities are multiplied by $v'[k]$, so that the total of the probabilities for the row is k. This finishes the derivation.

To construct the matrix, we modify the values in two ways. Since the node degrees have an exponential distribution, the values of the bottom rows are often much larger than the other degrees. The result for values of k and l close to n can be greater than one. To avoid having elements of the matrix with values greater than one (i.e. probability greater than one), we compute the matrix elements in a double loop starting with k (outer loop) and i (inner loop) set to n-1. The values of k and i are decremented from n-1 to zero. If the value is greater than one then the corresponding element is set to one and the value copied from $v'[k]$ for computing row k is decremented by one. This keeps all matrix elements in the range zero to one, so that they may represent valid probabilities.

The other modification of element values forces the matrix to be symmetric. When computing a row k and $k < n-1$, all elements for $i > k$ are set to be the same as the values computed for element (i, k). If the value of element (i, k) is one, the value copied from $v'[k]$ is again decremented. In some cases this may force the sum of row k to deviate from $v'[k]$.)If the deviation is significant enough, the resulting connectivity matrix may only have a degree distribution that approximates the scaling factor γ.) An example connectivity matrix for n=10 and γ=2.0 is:

$$\begin{bmatrix}
0 & 1/22 & 1/22 & 1/22 & 1/22 & 1/22 & 1/10 & 1/10 & 2/9 & 9/10 \\
1/22 & 0 & 1/22 & 1/22 & 1/22 & 1/22 & 1/10 & 1/10 & 2/9 & 9/10 \\
1/22 & 1/22 & 0 & 1/22 & 1/22 & 1/22 & 1/10 & 1/10 & 2/9 & 9/10 \\
1/22 & 1/22 & 1/22 & 0 & 1/22 & 1/22 & 1/10 & 1/10 & 2/9 & 9/10 \\
1/22 & 1/22 & 1/22 & 1/22 & 0 & 1/22 & 1/10 & 1/10 & 2/9 & 9/10 \\
1/22 & 1/22 & 1/22 & 1/22 & 1/22 & 0 & 1/10 & 1/10 & 2/9 & 9/10 \\
1/10 & 1/10 & 1/10 & 1/10 & 1/10 & 1/10 & 0 & 1/5 & 4/9 & 1 \\
1/10 & 1/10 & 1/10 & 1/10 & 1/10 & 1/10 & 1/5 & 0 & 4/9 & 1 \\
2/9 & 2/9 & 2/9 & 2/9 & 2/9 & 2/9 & 4/9 & 4/9 & 0 & 1 \\
9/10 & 9/10 & 9/10 & 9/10 & 9/10 & 9/10 & 1 & 1 & 1 & 0
\end{bmatrix}$$

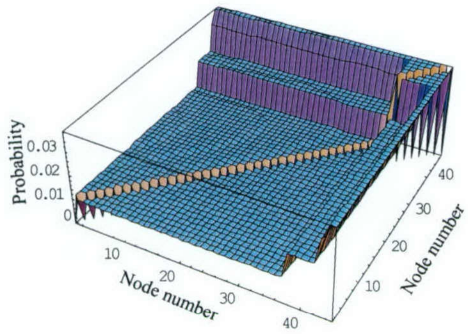


Figure 24. Three-dimensional plot of the connectivity matrix for a scale free graph with n=45 and γ=3.0. Note the zero diagonal and the high probability of connections to the hub nodes. Connections between hub nodes are virtually assured. Connections between non-hub nodes are very improbable.

Scale-free networks provide a good statistical description of large, evolving, wired networks with no centralized control. Mobile wireless networks are also of importance. In particular *ad hoc* wireless networks, which have no fixed infrastructure, are suited to analysis as a type of random graph. [Krishnamachari 2001] explains a fixed radius model for random graphs that they use to analyze phase change problems in ad hoc network design.. The model places nodes at random in a limited two-dimensional region. Two uniform random variables provide a node's x and y coordinates. Two nodes in proximity to each other have a very high probability of being able to communicate. For this reason, they calculate the distance r between all pairs of nodes. If r is less than a given threshold, then an edge exists between the two nodes. In their work, many similarities are found between this graph class and the graphs studied by Erdös and Rényi. Their analysis looks at finding phase transitions for constraint satisfaction problems. These graphs differ from Erdös-Rényi graphs in that they have significant clustering. We will use this model, except that they create an edge with probability one when the distance between two nodes is less than the threshold value. We will allow the probability to be set to any value in the range [0..1]. Figure 6 shows an example range limited random graph.

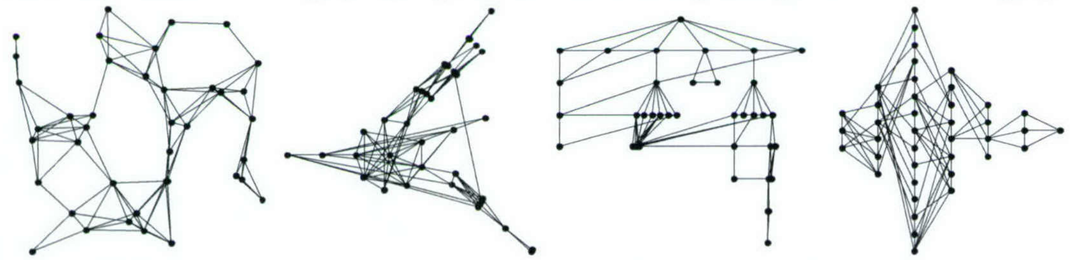


Figure 25. Different embeddings of a range limited random graph of 40 nodes positioned at random in a unit square region. The distance threshold was set as 0.25, and within that range edges exist with a probability of one. From left to right: geographic locations, radial embedding, rooted embedding with node 40 as the root, and ranked embedding from nodes 38, 39, and 40.

We construct range-limited graphs from the following parameters:

n – the number of nodes

max_x (max_y) – the size of the region in the x (y) direction

r – the maximum distance between nodes where connections are possible

p – probability that an edge exists connecting two nodes within the range

Construction of range-limited random graphs proceeds in two steps: (*i*) sort the nodes by either their x (or possibly y) coordinate and use order statistics to find the expected values of that coordinate, (*ii*) determine probabilities for edges existing between nodes based on these expected values.

To construct the connectivity matrix for range-limited graphs, we consider the position of each node as a point defined by two random variables: the x and y location. Without loss of generality, we use normalized values for the x, y, and r variables limiting their range to [0,1]. To calculate probabilities, we sort each point by its x variable. For the n nodes, rank statistics provide expected value $j/(n+1)$ for the node in position j in the sorted list. Using Euclidean distance, an edge exists between two nodes j and k with probability p when:

$$(x_j - x_k)^2 + (y_j - y_k)^2 \le r^2$$

By entering the expected values for nodes of rank j and k and re-ordering terms, this becomes:

$$(y_j - y_k)^2 \le r^2 - \left(\frac{j}{n+1} - \frac{k}{n+1}\right)^2$$

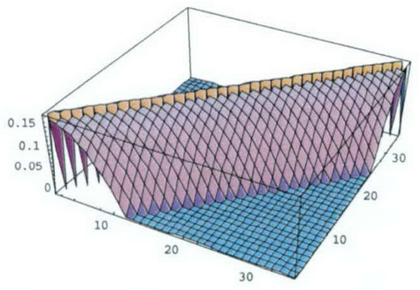


Figure 26. Three-dimensional plot of the connectivity matrix for a range limited graph of 35 nodes with range of 0.3.

If we assume that the random variables giving the x and y positions are uniformly distributed and uncorrelated, the probability that the relation holds is the probability that the square of the difference of two normalized uniform random variables is less than the constant value c provided by the right hand side. Two uniform random variables describe a square region, where every point is equally likely. The equation is an inequality, so it defines a closed linear region. Because the right hand side is squared, two symmetric regions are excluded from the probability. The limiting points are when y_j or y_k are equal to the constant on the left hand side. Algebraic manipulation provides the equation $2c\text{-}c^2$ for the probability. An example matrix for six nodes in a unit square with r=0.3 and p=1.0 is:

$$\begin{bmatrix} 0 & 0.134 & 0.0167 & 0 & 0 & 0 \\ 0.134 & 0 & 0.134 & 0.0167 & 0 & 0 \\ 0.0167 & 0.134 & 0 & 0.134 & 0.0167 & 0 \\ 0 & 0.0167 & 0.134 & 0 & 0.134 & 0.0167 \\ 0 & 0 & 0.0167 & 0.134 & 0 & 0.134 \\ 0 & 0 & 0 & 0.0167 & 0.134 & 0 \end{bmatrix}$$

Figure 26 shows a three-dimensional plot of an example matrix.

We have illustrated how to construct these matrices for important graph classes. We now discuss the structure and meaning of the matrices. By definition, connectivity matrices are square with the numbers of rows and columns both equal to the number of vertices in the graph (n). Each element (j,k) is the probability of an edge existing between nodes j and k. Since we consider only non-directed graphs, (j,k) should equal (k,j). (Care should be taken to guarantee that algorithms for constructing matrices provide symmetric results.)

Remark. The sum of each row (column) of the probabilistic connectivity matrix M provides the expected degree of every node in G.

Definition. M is a probabilistic connectivity matrix of random graph class G. We define *rpm*-graphs (*r*egular *p*robability *m*atrix) graphs as graph classes where each node has equivalent probabilities of being connected to other nodes.

Proposition: *If a graph class defines rpm-graphs, all nodes in G are equivalent and every row (column) is a permutation of every other row (column).*

Proof. [Cvetovic 79] provides a proof that each row of the connectivity matrix of a *regular* graph is a permutation of every other row; this proof suffices for *rpm*-graphs.

Proposition. *For an rpm-graph class G, the value of the eigenvalue with the largest magnitude of its associated connectivity matrix is the expected degree of each node in G.*

Proof. This proof follows directly from the proof of this assertion for non-random graphs given in [Cvetovic 79].

Recall that each element (j,k) of the random connectivity matrix is a probability and constrained to values between zero and one. (It represents the likelihood of an edge existing between nodes j and k.) The product of two probabilities values gives the likelihood of their two associated events occurring together when independence is assumed. So the likelihood of edges existing simultaneously from node j to node k and node k

to node l, is (j,k) (k,l). As we have shown in constructing the matrices for scale-free and small-world graphs, we can construct probability matrices where the values of the probabilities explicitly state the influence of statistical dependencies.

Now consider the likelihood of a path of length two existing between nodes j and l. This can be calculated by summing the values of (j,k) (k,l) for all possible intermediate nodes:

$$\sum_{k=1}^{n}(j,k)(k,l)$$

Note that this is the equation used in matrix multiplication to calculate element (j,l) of M_1M_2 by multiplying row j of M_1 by column l of M_2. We discuss further applications of this in section 4.1 to 4.3. Consider computation of this value. The values used are probabilities with an assumption of independence and results need to be probabilities as well. Numerically the probability of either of two independent events j and k occurring is: $P_j + P_k - P_jP_k$. The probability of three events j, k, and l occurring can be computed recursively as $P_l + (P_j + P_k - P_jP_k) - P_l(P_j + P_k - P_jP_k)$. As the number of events increases the number of factors involved increases, making this computation awkward for large matrices. An equivalent computation is:

$$1-\prod_{k=1}^{n}\left(1-P_{jk}P_{kl}\right)$$

This is easier to compute and suggested for the matrix multiplications discussed in the application sections.

As a matter of convention, the diagonal values (P_{jj}) of connectivity matrices can be set either to one or zero. Frequently they are set to one, signaling implicitly that each graph vertex is connected with itself. Which is often reasonable. For this approach, the diagonal value of zero more appropriate. Our applications concern the likelihood of paths existing between nodes. The value P_{jj} expresses the probability of a path connecting node j with itself. The existence of a loop within the graph should not increase the probability that two nodes in the graph are connected. Constraining the diagonal values to the value zero discounts the influence of loops in our calculations. For some random graph classes, like Scale-Free graphs, it is advisable to add a post-processing step to multiplication where each element (j,k) is set to the maximum of (j,k) and (k,j) to guarantee that the matrix remains symmetric.

Theorem. *Element (j,k) of M^z is the probability that a walk of length z exists between nodes j and k.*

Proof. The proof is by induction. By definition, each element (j,k) is the probability of an edge existing between nodes j and k. M^2 is the result of multiplying matrix M with itself. Equation (12) is used to calculate each element (j,k) since all values are probabilities. As explained in section 4, this calculates the probability of a path of length two existing between nodes j and k by exhaustively enumerating the likelihood of the path passing through each intermediate node in the graph. Using the same logic, M^z can be calculated from M^{z-1}using matrix multiplication to consider all possible intermediate nodes between nodes j and l. Where M^{z-1} has the probabilities of a walk of length z-1 between j and k, and M has the values defined previously.

Example 1. Probabilities of walks of length three in an Erdös-Rényi graph of four nodes for p=0.6 and 0.65

$$M=\begin{bmatrix}0&0.65&0.65&0.65\\0.65&0&0.65&0.65\\0.65&0.65&0&0.65\\0.65&0.65&0.65&0\end{bmatrix}\quad M^2=\begin{bmatrix}0&0.666&0.666&0.666\\0.666&0&0.666&0.666\\0.666&0.666&0&0.666\\0.666&0.666&0.666&0\end{bmatrix}\quad M^3=\begin{bmatrix}0&0.679&0.679&0.679\\0.679&0&0.679&0.679\\0.679&0.679&0&0.679\\0.679&0.679&0.679&0\end{bmatrix}$$

$$M=\begin{bmatrix}0&0.6&0.6&0.6\\0.6&0&0.6&0.6\\0.6&0.6&0&0.6\\0.6&0.6&0.6&0\end{bmatrix}\quad M^2=\begin{bmatrix}0&0.59&0.59&0.59\\0.59&0&0.59&0.59\\0.59&0.59&0&0.59\\0.59&0.59&0.59&0\end{bmatrix}\quad M^3=\begin{bmatrix}0&0.583&0.583&0.583\\0.583&0&0.583&0.583\\0.583&0.583&0&0.583\\0.583&0.583&0.583&0\end{bmatrix}\qquad(14)$$

Many graph properties follow a 0-1 law. The property will either appear with probability of near 0 or probability of near 1 in a random graph class as the graph size increases, depending on the parameters that define the class. Frequently, an abrupt phase transition exists between these two phases [13, 2]. The parameter value where the phase transition occurs is referred to as the critical point. The connectivity matrices defined in this paper can be useful for identifying critical points and phase transitions.

Theorem. *For Erdös-Rényi graphs of n nodes and probability P of an edge existing between any two nodes, the critical point for the property of graph connectivity occurs when* $P = 1-\left(1-P^2\right)^{n-1}$. *When* $P > 1-\left(1-P^2\right)^{n-1}$, *the graph will tend to not to be connected. When* $P < 1-\left(1-P^2\right)^{n-1}$, *the graph will tend to be connected.*

Proof. For Erdös-Rényi graphs, all non-diagonal elements of the matrix have the same value p. Diagonal elements have the value zero. The formula $1-(1-P^2)^{n-1}$ follows directly from these two facts and (13). When the value of this equation is equal to p, two nodes are just as likely to have a two-hop walk between them as a single edge. This means that connections of any number of hops are all equally likely. When the value of the equation is less than p, a walk of two hops is less probable than a single hop connection. Since the equation is monotonically decreasing (increasing) as p decreases (increases). This means that longer walks are increasingly unlikely and the graph will tend not to be connected. By symmetry when the value of the equation is greater than p, the graph will tend to be connected.

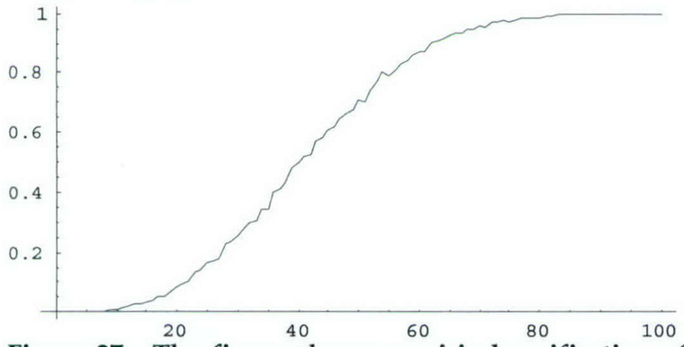


Figure 27. The figure shows empirical verification of theorem for Erdös-Rényi graph connectivity. Two thousand instances of Erdös-Rényi graphs of five nodes were generated as the edge connection probability varied from 0.01 to 1.00. The *x*-axis times 0.01 is the edge probability. The *y*-axis is the percent of graphs that were connected. The formula used in the theorem predicts the critical value around probability 0.4. When *p*= 0.35 (0.40) equation (16) gives 0.357 (0.407).

4.3 TRUST PROPAGATION IN P2P NETWORKS

This section provides a set of key distribution protocols that can be used for secure mobile code package transfer using this general structure. Both secret (symmetric) and public (asymmetric) protocols are given. These protocols are typical of those documented in the literature and in use today.

These protocols describe all the steps that would be typically used to set up a connection and exchange mobile code packages. Expected values for the message sizes can be found for any specific instance of a P2P network, like the one we propose. The design parameters for packet time outs and number of indexes can then be computed.

Out topology currently supports two separate trust models. Both models assign to the index the role of key server and guardian of the trustworthiness of the nodes they serve. They differ in how the trustworthiness of the index nodes is verified.

One trust model is consistent with current Public Key Infrastructure (PKI) design. Each node is considered trustworthy based on its verification by a higher authority. In which case, trust propagation takes the form of a directed acyclic graph (DAG). The highest level of the hierarchy is an entity, which is assumed inviolate and always trustworthy. In our terminology, this entity is God.

The alternative topology does not assume the existence of an inviolate entity. It does assume that fewer than 1/3 of the currently active indexes can be corrupted at any point in time. In this case, indexes can be removed from the system by the other indexes using a Byzantine Generals approach [Brooks 1998].

4.3.1. PUBLIC KEY INFRASTRUCTURE

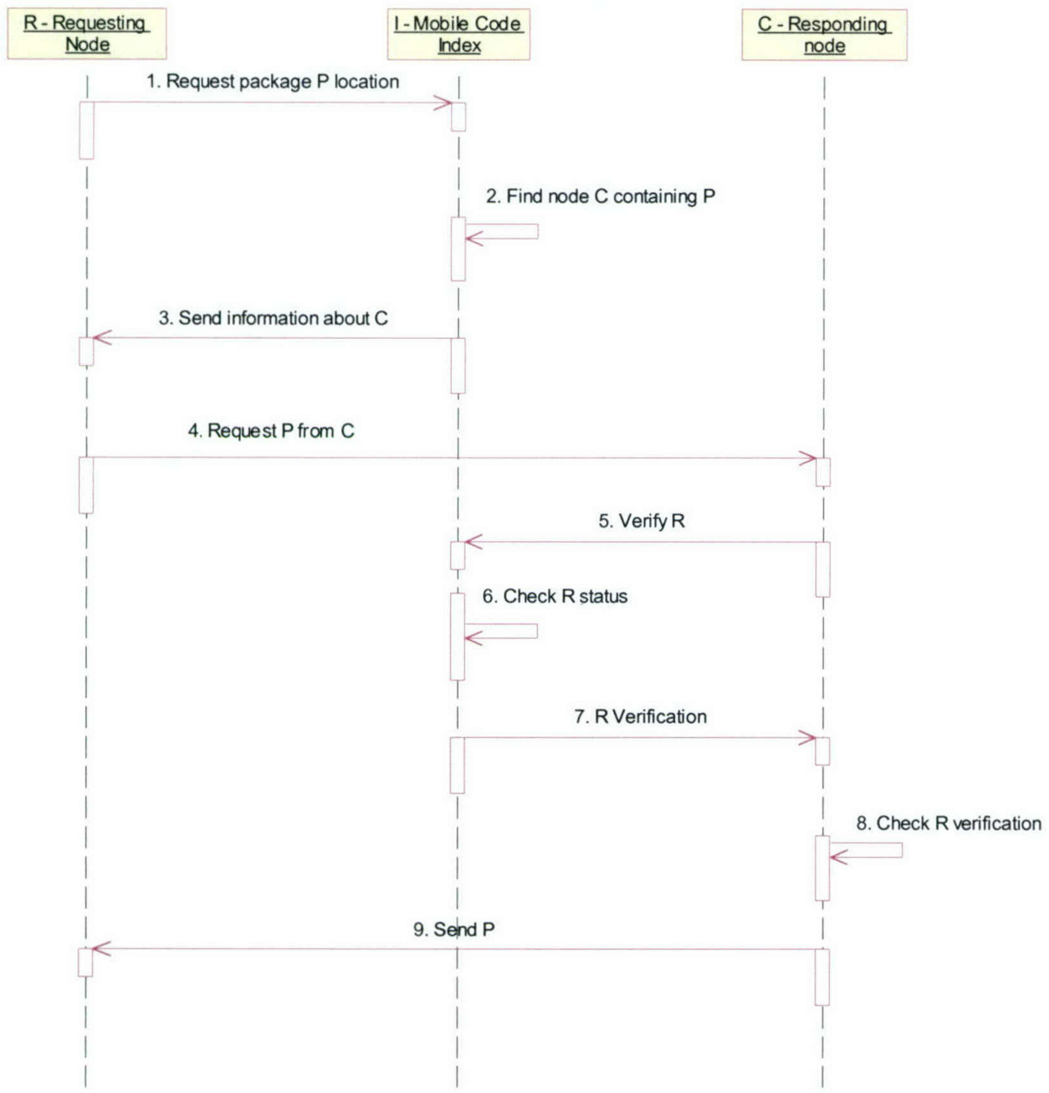


Figure 28. Public Key Protocol

IU is public key of Index I
IR is private key of Index I
RU is public key of Node R
RR is private key of Node R
CU is public key of Node C
CR is private key of Node C
Message 1:

Node R sends a message asking Index I where mobile code package P is.

Request contains:

1. **Name of P (Encrypted)**
2. **Nonce (Encrypted)**
3. **Address of R's index if I is not R's index**

4. R's Address (Clear text)

Both 1 and 2 are encrypted with R's private key RR
The entire request is encrypted using Index I's public key IU.

The Nonce is undefined but may contain:

1. Request sequence number
2. Time stamp
3. Random number

or some combination of the above

Message 2:

I decrypts the message with its private key. Verify whether R's privileges have been revoked. If so, then drop the request.

If R is not a node normally served by I, checking for revocation of R requires checking that the privileges of the index given in 1 have not been revoked. And then checking with that index that R has not been revoked. If either one fails, drop the request.

R's public key - RU - is known by I if R is normally served by I. Else RU is retrieved from R's index while checking for revocation of R's privileges.

I decrypts the name of P using R's public key to verify that the message is from R. Optionally check validity of the nonce and drop the request if it is invalid. Find address of node C containing package P. If package P can not be found address of C is Null.

Message 3:

I sends to R:

1. Network address of node C containing package P
2. Public Key CU of node C
3. Nonce from message 1

This message is encrypted with Private Key of Index - IR and Public Key of R – RU.

R decrypts the message. Verifies the source of the message and that the nonce matches the nonce in message 1. If the address of C is NULL, then the protocol recommences at the next index. R can also time-out while waiting for this message. If it does so and not all indexes have been checked, then the protocol recommences at the next index. If R times out and all indexes have been checked, then the request fails. If the address of C is non-NULL and R has not timed out, then the protocol continues.

Message 4:

Node R requests package P from node C by sending:

1. Nonce from message 1
2. Address of R
3. RU in plain text
4. Name of package P
5. Hash code of 1-4 encrypted with RR

This message encrypted is with C public Key - CU. After decrypting with CR, C does hash of 1-4 and compares with 5 decrypted with RU. This verifies that R is the message source and that the message has not been modified.

Message 5:

Message sent to Index I containing:

1. Address of R
2. R Public Key - RU
3. Address of C
4. Nonce

All are encrypted using IU.

Message 6:

Decrypt using IR. Look to verify that node R's access has not been revoked. Since verification of nodes not served by R was done by message 2, no extra processing for that case is necessary here. (I retains the fact that R was valid for a limited time). If R has been revoked set R verification to NULL. Else set R verification to True. If R verification is true, check as well that RU is in fact the public key of R.

Message 7:

I sends message to C containing:

1. Nonce from 5
2. R verification
3. R Address

Both are encrypted using IR and CU

Message 8:

Decrypt using CR. Then C verifies that 7 came from I using IU. Verify which package P is needed using Nonce. If R is still part of the system prepare to send P. Else drop the processing.

Message 9:

C sends message to R containing:

1. Nonce from step 4
2. P
3. Hash of 1 and 2 encrypted using CR

The message is encrypted with RU. R may have timed out during the process. If all indexes have been tried and timed out then the request fails. If R times out and an index remains, the protocol recommences using the next index. If R receives message 9. It decrypts the message and does the following:

1. **It uses the nonce to match this with the correct request.**
2. **R compares the hash of 1 and 2 with 3 decrypted using CU. To verify the source and integrity of 9.**
3. **The request terminates.**

4.3.2. Modifications of existing symmetric key protocols for P2P exchange of mobile code packages

For all symmetric key protocols:

KR is a symmetric key known only to R and I
KC is a symmetric key known only to C and I

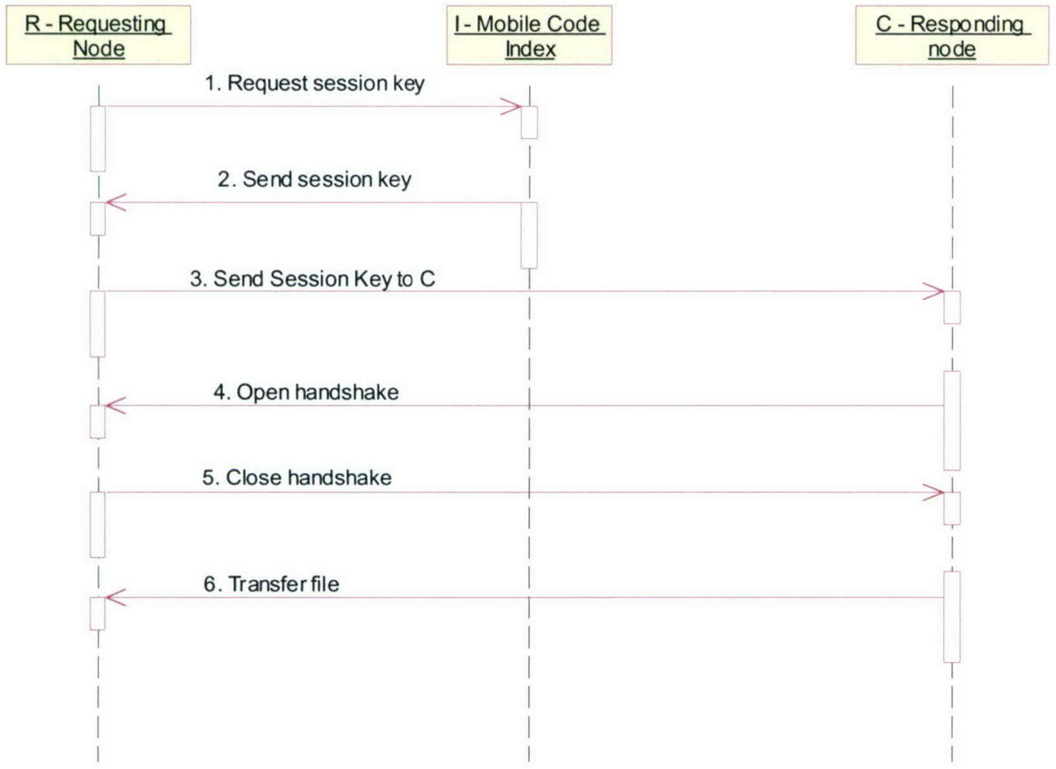


Figure 29. Needham-Schroder Secret Key protocol

Message 1:

Node R sends to I:

1. Address of R
2. Name of Package P
3. Nonce

(In the classical Needham-Schroeder the requester sends the address of node C instead of Package Name).

Message 2:

I sends to R:

Encrypted with KR{

1. Nonce from message 1.
2. Address of C
3. Name of P
4. Symmetric Session Key in Clear Text
5. Encrypted with KC{

1. Session Key in clear text
2. Address of R

}

}

(In the classic Needham-Schroeder Protocol, item 3 is not needed).

Message 3:

R sends to C:

Encrypted with KC{

  1. Session Key in clear text

  2. Address of R

}

Message 4:

C sends a nonce to R encrypted with the symmetric key

Message 5:

R sends to C:

Encrypted with session key{

  1. Nonce-1

  2. Program module name

}

Message 6:

C sends to R:

Encrypted with session key{

  1. Mobile code module P

  2. Nonce-2

}

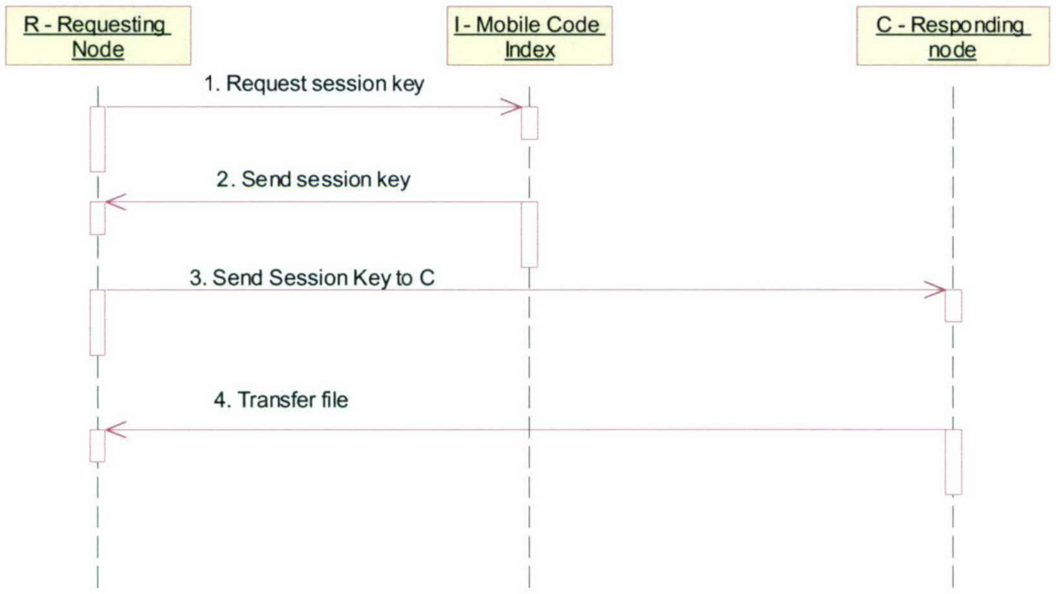


Figure 30. Denning Sacco Secret Key protocol

Message 1:

Node R sends to I:

  1. Address of R

  2. Name of Package P

(In the classical Denning-Sacco the requester sends the address of node C instead of Package Name).

Message 2:

I sends to R:

```
Encrypted with KR{
        1. Time stamp
        2. Address of C
        3. Name of P
        4. Symmetric Session Key in Clear Text
        5. Encrypted with KC{
                1. Session Key in clear text
                2. Address of R
                3. Time Stamp
        }
}
(In the classic Needham-Schroeder Protocol, item 3 is not needed).
```

Message 3:

```
R sends to C:
        1. Encrypted with KC{
                1. Session Key in clear text
                2. Address of R
                3. Time stamp
        }
        2. Program module name encrypted with Session Key
```

Message 4:

```
C sends to R:
Encrypted with session key{
        1. Mobile code module P
}
```

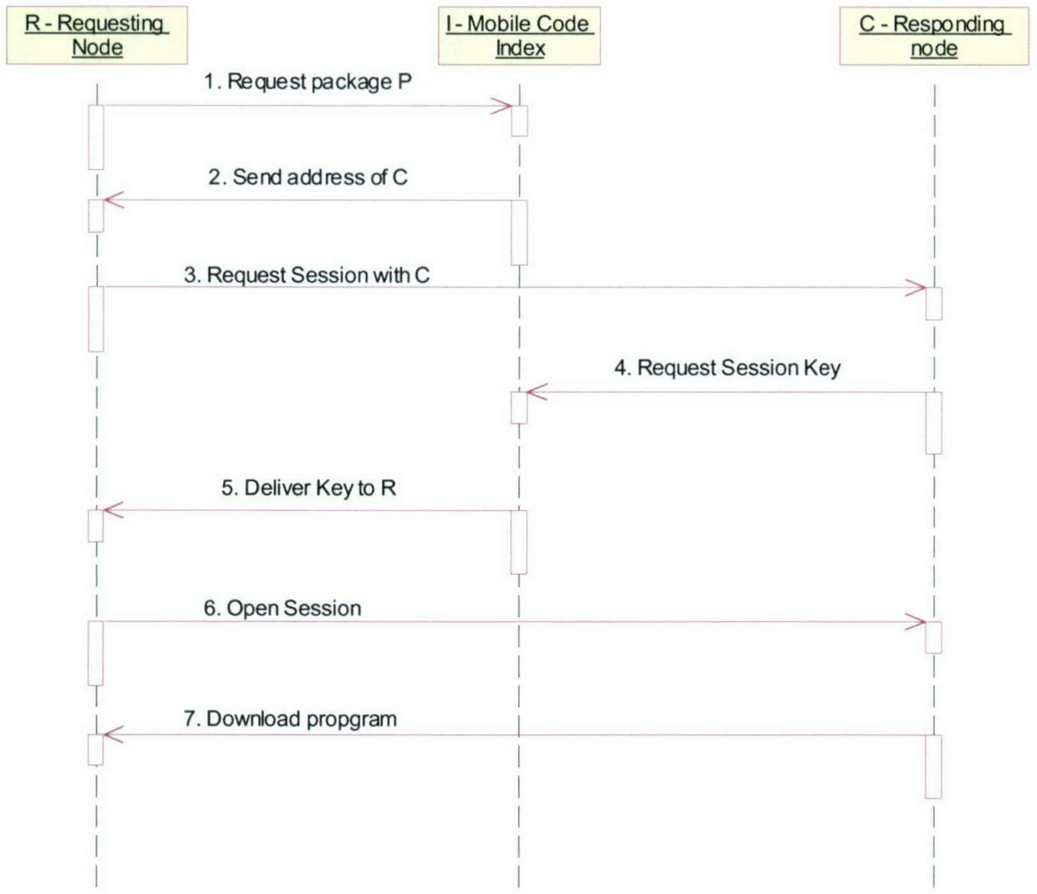


Figure 31. Yaholom Protocol

Message 1:

Node R sends to I:

1. **Address of R**
2. **Name of Package P**
3. **Nonce**

(Not in classic Yahalom)

Message 2:

I sends to R:

Encrypted with KR{

1. **Nonce from message 1.**
2. **Address of C**

}

(Not in classic Yahalom)

Message 3:

R sends to C:

1. **Nonce R**
2. **Address of R**

```
          (Start of Yahalom)

Message 4:
          C sends to I:
          1. Address of R
          2. Encrypted with KC{
                    1. Address of A
                    2. Nonce R
                    3. Nonce C
          }

Message 5:
          I sends to R:
          1. Encrypted with KR{
                    1. Address of C
                    2. Session Key
                    3. Nonce R
                    4. Nonce C
          }
          2. Encrypted with KC{
                    1. Address of R
                    2. Session Key
          }

Message 6:
          R sends to C:
          1. Encrypted with KC{
                    1. Address of A
                    2. Session Key
          }
          2. Encrypted with session key{
                    1. Nonce C
                    2. Name of P
          }

Message 7:
C sends to R:
Encrypted with session key{
          1. Mobile code module P
}
```

4.4 NODE DESIGN AND IMPLEMENTATION

The mobile code daemon we present is based upon a core network protocol called the Remote Execution and Action Protocol (REAP). This protocol is responsible for message passing between nodes within our network. On top of this packet protocol we have developed a framework to allow objects to serialize themselves and travel across the network. At a higher layer of abstraction we have written messages to handle remote process creation and monitoring, simple file system operations, and resource index operations.

We have tested the daemons in two distributed applications. The first application is a battery-powered wireless sensor network. Nodes are fielded with a minimal software suite that can be extended as required using mobile code. In particular, we showed the utility of changing target classification modules dynamically based on the set of targets observed in the environment. Use of the mobile code daemon in sensor network applications is documented in [Brooks 2000].

The second application is a dynamic battle management scenario. A distributed process, such as target allocation, consists of multiple tasks that may be performed at different locations. The enemy needs to disrupt the target allocation process, and uses Denial of Service (DoS) attacks to disrupt the processing pipeline. The system uses the mobile code daemon for introspection and re-allocates tasks to new nodes as required. Once again, mobile code is seen to be useful in adapting to a chaotic environment.

The mobile code daemon framework is built on the mobile code taxonomy given in [Orr 2002]. This taxonomy shows that the established mobile code paradigms of client server, code on demand, remote evaluation, and mobile agents [Brooks 2002] can all be expressed using a single abstraction. A mobile code system is defined by a distributed behavior. The behavior is a set of message transmissions between cooperating nodes that follow an itinerary. Messages may consist of programs, data, execution commands, resource allocation requests, etc. To cooperate, the nodes need to offer a local execution environment that remote nodes can access [Orr 2002].

The daemon presented here provides a common execution environment, somewhat analogous to a Java Virtual Machine. Unlike the Java Virtual Machine, the REAP protocol can mimic any of the established mobile code approaches. It can also = create applications that do not fit the common paradigms.

The daemon is written in C++. The first version ran on the Windows NT and Windows CE operating systems. It has since been ported to the Linux operating system. The daemon structure is broken down into several core modules: foundation classes, the networking core, the random graph module, the messaging core, the packet router, the index server, the transaction manager, the resource manager, and the process manager. We will discuss each of these components in turn.

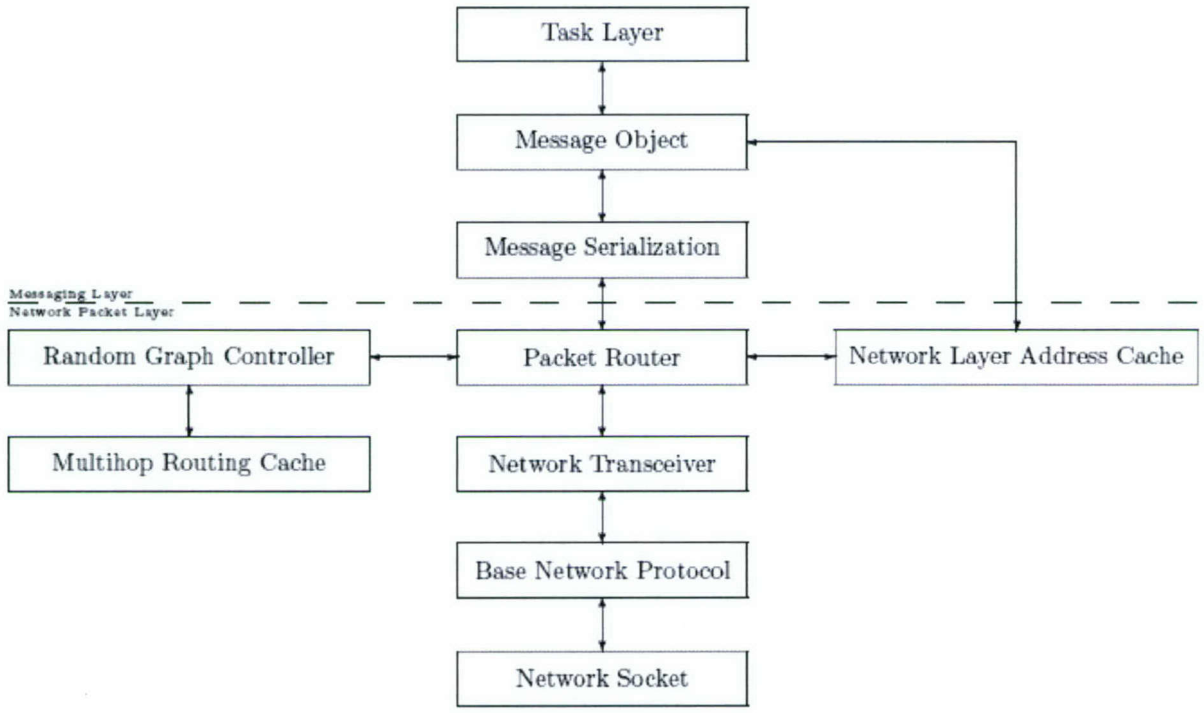


Figure 32. Daemon structure.

Before discussing the REAP daemon in detail, it is useful to discuss its underlying framework on which it is built. The framework abstracts many of the complexities of systems programming out of the core, into a set of libraries. Thus, we have written our own object-oriented threading and locking classes, whose current implementation calls into the threads library of the underlying operating system. We also rely heavily on a set of templated, multithreaded linked list, hash, and heap objects throughout the code. In addition, there are classes to handle singleton objects, the union-find problem, and object serialization. Lastly, there is also a polymorphic socket library that allows different networking architectures to emulate unicast stream sockets, regardless of the underlying network protocol or topology. These socket libraries are explained in the discussion of the networking core.

The daemon is capable of communicating over several networking technologies. The major ones are: TCP/IP, Diffusion Routing, and UNIX domain sockets. The socket framework is designed so that new protocols are easily inserted into the daemon. To achieve this, an abstract base class Socket includes all of the familiar calls to handle network I/O. Furthermore, all nodes are assigned a protocol-independent unique address. Opening a new socket involves looking up the network-layer address of a node in a local cache, and then opening the lower-level socket. When a cache miss occurs, a higher-level protocol is provided to find the network-layer address. The appropriate socket object is allocated based upon the network-layer address of the destination.

Diffusion provided some interesting challenges because it is not a stream-oriented unicast protocol. Rather, it provides a publish and subscribe interface, and is essentially a multicast datagram protocol. Thus, we had the choice of rewriting the REAP socket protocol as a datagram protocol, or building a reliable stream protocol on top of the Diffusion framework. It was deemed simpler to write a reliable stream protocol on top of Diffusion. In essence, we wrote a simplified userspace TCP stack. The current userspace stack employs the standard three-way handshake protocols for socket open and close, and it also employs a simple delayed-ACK algorithm. This system is implemented as an abstract child of the Socket base class. Our Diffusion driver then provides an implementation of our userspace TCP module. The Diffusion driver performs a role equivalent to the IP layer processing code in most kernels. It receives datagrams from the Diffusion daemon through callback functions, parses the headers to make sure the datagram has reached the correct destination, and then either discards the contents, or passes it up to the TCP layer. These steps were deemed necessary because Diffusion is a multicast protocol, and thus we could not rule out the possibility of datagrams reaching our socket object that were not actually destined for it.

Early on in the project, it became clear that persistent connections between the various nodes was essential. A single file transfer of a shared object could result in thousands of packets traversing the network, and session setup time was simply too long over TCP and Diffusion. To counteract this problem we implemented a system whereby sockets are kept open whenever possible. The first implementation of this system opened directly to a destination, and did not support multi-hop routing very well. Under this implementation, socket timeout counters were employed to close underutilized sockets. This method has inherent scalability problems, and we decided a better solution was required.

Offset	Word Contents		
0	Protocol	Version	Command Code
4	Source Node ID		
8	Source Process ID		
12	Source Task ID		
16	Source Ticket ID		
20	Destination Node ID		
24	Destination Process ID		
28	Sequence Number	Max Sequence	
32	Packet Size	Options	TTL
	Extended Header (0–60 bytes)		
	Data (0–65495 bytes)		
	CRC Checksum		

Figure 32. REAP packet structure.

This better solution involves a multi-hop packet routing network built on top of a random graph of sensor nodes. Each node in the system has four graph parameters specified: minimum degree, maximum degree, cliquishness, and clique radius. The cliquishness parameter defines the probability of a new edge being formed to a node within the clique radius. The minimum degree and maximum degree parameters control how many neighboring nodes can exist at any point in time. The clique parameters allow us to control the size and connectedness of cliques within the graph. Cliques become more important when we investigate the index system.

To add a new edge, a random number is generated to decide whether or not to add a clique edge. Then a random node from the node cache is chosen based upon two filter criteria: the chosen node must have a minimum path length of two to this node, and its minimum path length must be less than or equal to the clique radius for a clique edge, or greater than the clique radius for a non-clique edge.

The messaging system implements the core of the REAP protocol. At its lowest levels, this consists of a packet protocol, on top of which serialized objects are built. The Packet class is nothing more than a variable-sized opaque data carrier that is capable of sending itself between nodes, and it also performs data and header checksumming. The layout of a REAP packet is shown in Figure. The header defines enough information to route packets, specify the upper-level protocol, and to handle multi-packet transmissions where the number of packets is known a priori. The options field consists of a 4-bit options vector, and a 4-bit header extension size parameter. The TTL field is used in the new multi-hop protocol to eventually destroy any packet routing loops that might form.

Higher level messaging functionality is handled by a set of classes that do object serialization, and by a base message class. The serialization class in REAP provides a fast method of changing common data types into network byte-ordered, opaque data. The key advantage to this serialization system is that it only handles common data types, and thus has much lower overhead than technologies such as XDR and ASN.1.

The base messaging class provides a simple interface to control destination address, source transaction information, possible system state dependencies for message delivery, and control over sending the message. In addition, it defines abstract serialization and reordering functions that are implemented by all message types.

The serialization class sits beneath the base message class and does the physical work of serializing data, packetizing the serialized buffer, and then injecting those packets into the router.

On the receiving end, packets are received by an object serialization class and inserted into the proper offset in the receive buffer. A union-find structure keeps track of packet sequence numbers, and once it detects that all packets have been received, the message is delivered to a message queue in the destination task structure.

Another interesting feature of the messaging system is the function called run. This function takes a task structure as an argument, and is generally intended to perform some action on the destination of the message. We will see an example of this later on when we discuss the index server.

The daemon packet router has several key responsibilities. The primary one is to use its internal routing tables to move packets from source to destination. The other primary function of the router is to coordinate the dissemination of multi-hop routing data.

The current method of determining multi-hop paths is through broadcast query messages. We gradually increase the broadcast TTL until a route is found, or a TTL upper limit is reached, at which point the node is assumed down. This methodology helps to reduce flooding, while making optimal paths likely to be found. A simple optimization allows a node to answer a multi-hop query if it has an answer in its routing table. Although this system is essentially a heuristic, it tends to work well because failed intermediate nodes are easily bypassed when their neighbors find that they cannot reach the next hop. Of course, this can lead to much longer paths through the graph, but support is integrated to warn of intermediate node failures, and multi-hop cache expire times help to reduce this problem by forcing refreshes occasionally. The multi-hop refreshes are carried out in unicast fashion, and a broadcast refresh is only used if a significant hop count increase is detected.

The actual routing of packets involves looking at the two destination fields in the packet header. First, a check is performed to determine whether the destination node identifier is equivalent to the current node's identifier, or the local loopback address, or one of several addresses that are defined for special purposes, such as broadcast to all members of a clique. The next check is to determine whether the destination process identifier is equivalent to that of the current process. If it is not, then the packet will need to be forwarded across a unix domain socket. If both of these tests pass, then the packet must be delivered to the appropriate task. Because packets do not contain sufficient routing data to deliver them to a specific task, we must recreate the high level message object in the router to determine the message's final destination.

Every task in a REAP process registers itself with the router during initialization. Once a task is registered, it can receive messages bound for any active ticket. Several special tickets are defined for every task that handle task status messages, and task-wide requests. Other tickets are ephemeral, and are allocated as needed.

An important component of the REAP daemon is the index system. This system implements a distributed database of resource available on the network. Each record in this database describes an object of one of the following types: index server, file, executable, library, pipe, memory map, host, or a task. Every record in the database has a canonical name, and resource locator associated with it. Both of these values are stored as human-readable strings. Besides this, metadata to allow for both data and metadata replication are present. The goal is to have a distributed cluster of index servers that transparently replicate each other's index records, and to have a resource control system that transparently replicates the actual data as well. At this point, the replication technology is only partially implemented.

The index system consists of the following modules: client, server, database, and the associated messaging protocol. The client is responsible for building a query message, sending the message, and either waiting for a response, or returning a response handle to the client in the case of an asynchronous call. The server consists of a pool of threads that poll for incoming messages on the server task structure. When a thread receives a message, it runs the query embedded in the message against the local database, and then sends the results back to the client in a query result message.

$$
\begin{array}{rcl}
G & \rightarrow & \text{OP\_ATOMIC}(A,A) \mid \text{OP\_NOP}(G,G) \mid \text{OP\_END}(G,G) \mid \text{OP\_IF}(B,G) \mid \\
 & & \text{OP\_ELSE}(B,G) \mid \text{ACT\_SET}(F,X) \mid \text{ACT\_INC}(F) \mid \text{ACT\_DEC}(F) \mid \\
 & & \text{ACT\_REMOVE} \mid \text{ACT\_ADD}(R) \mid \text{CAST\_VOID}(B) \\
 & & \\
A & \rightarrow & \text{OP\_NOP}(A,A) \mid \text{OP\_END}(A,A) \mid \text{OP\_IF}(B_A,B_A) \mid \text{OP\_ELSE}(B_A,B_A) \mid \\
 & & \text{ACT\_SET}(F,X) \mid \text{ACT\_INC}(F) \mid \text{ACT\_DEC}(F) \mid \text{ACT\_REMOVE} \mid \\
 & & \text{ACT\_ADD}(R) \mid \text{CAST\_VOID}(B) \\
 & & \\
B & \rightarrow & C \mid \text{OP\_AND}(B,B) \mid \text{OP\_OR}(B,B) \mid \text{OP\_XOR}(B,B) \mid \text{OP\_NOT}(B) \mid \text{OP\_XOR}(B,B) \mid \\
 & & \text{CAST\_TRUE}(G) \mid \text{CAST\_FALSE}(G) \\
 & & \\
B_A & \rightarrow & C \mid \text{OP\_AND}(B_A,B_A) \mid \text{OP\_OR}(B_A,B_A) \mid \text{OP\_XOR}(B_A,B_A) \mid \text{OP\_NOT}(B_A) \mid \text{OP\_XOR}(B_A,B_A) \mid \\
 & & \text{CAST\_TRUE}(A) \mid \text{CAST\_FALSE}(A) \\
 & & \\
C & \rightarrow & \text{OP\_EQ}(F,X) \mid \text{OP\_NEQ}(F,X) \mid \text{OP\_LT}(F,X) \mid \text{OP\_LE}(F,X) \mid \text{OP\_GT}(F,X) \mid \text{OP\_GE}(F,X) \\
 & & \\
F & ::= & type \mid classification \mid cname \mid url \mid locked \mid cacheable \mid cache\_expiration \mid \\
 & & meta\_min\_replicas \mid meta\_max\_replicas \mid meta\_replicas \mid meta\_prev \mid meta\_next \mid \\
 & & data\_min\_replicas \mid data\_max\_replicas \mid data\_replicas \mid data\_prev \mid data\_next \mid \\
 & & atime \mid ctime \mid mtime \mid platform \mid version \mid flags \mid idnum \mid opaque\_data \\
 & & \\
I & ::= & string \mid uint8 \mid uint16 \mid uint32 \mid blob \mid true \mid false \\
 & & \\
R & ::= & record\_reference \\
 & & \\
X & \rightarrow & F \mid I \mid R
\end{array}
$$

Figure 33. Query context free grammar for REAP.

The query system is based upon a fairly extensible parse tree. The context-free grammar for our query language is shown in Figure 33. The query language permits complex boolean filtering on most any variable defined in an index record. The index server is essentially a lightweight SQL server that is tailored to resource location.

The index infrastructure is mainly built upon two message types: a query message, and a result message. The query message consists of an operand tree, some query option flags, and possibly a list of index records. Once the query message reaches the server, it is received by a server thread, and the run function is called. This function performs a query against the index database object, and sends back a result message to the source node. Once these actions are complete, the run function returns, and then the index server deallocates the query object. The index server itself is nothing more than a pool of threads that accept a certain type of message, and then allow the messages to perform their actions. In this sense, the REAP messaging system implements the mobile agent paradigm.

The other major feature of the index system is a system to select code based upon destination system architecture and operating system. To handle this, system architecture and operating system are considered polymorphic class hierarchies. Every index record contains an enumeration defining its membership in each hierarchy. When a system requests object code or binary data, we must ensure that it is compatible with the destination system. Thus, every index query can filter based upon architecture, if desired. When a query indicates that architecture and/or operating system are a concern, then C++ dynamic_cast calls are made to ensure compatibility. Because we are using the C++ dynamic casting technology, supported architectures and operating systems are determined at compile time. It would not be a technically difficult modification to use human-readable strings, and runtime-defined polymorphic hierarchies. However, we chose the compile-time approach because it is faster, and the architectures and operating systems in our lab are relatively constant.

To give an example of how this technology would work, let's take an example of a sensor node having raw time series data that needs to be run through an FFT. Suppose a distributed process scheduler determines that it would be optimal to move the raw data to a wireless laptop that is deployed in the field. When the laptop goes to run the FFT, it queries the index database for a given FFT algorithm, and requests architecture polymorphic checking. Let's say this laptop has a processor with Intel's SSE and MMX extensions, but not the SSE2 extensions. When the index server processes the query, let's say it finds FFT algorithms that are compiled for 386, Pentium, SSE, Pentium 4, and Alpha EV5. When it filters these queries, it determines that it can cast the laptop into 386, Pentium, and SSE, but not Pentium 4 or Alpha EV5. The laptop will then attempt to download the optimal one, only dropping to slower implementations when it cannot download the fastest one.

All operations in REAP are addressed by their transaction address. This address consists of the 4-tuple $(node, process, task, ticket)$. These globally unique addresses permit flexible packet routing. A major goal of REAP is to permit network-wide interprocess communication through a simple high-level interface, without introducing high overhead. We will see how this goal is met when we discuss the resource management module of the REAP mobile code daemon.

In order to support the complex transaction routing system, a task control structure is required. All threads, and other major tasks have their own task structure. This structure is registered with the local packet router, and is where message structures get delivered. Its primary jobs are to handle message I/O, and to allocate tickets. Every active ticket has an associated incoming message queue, and thus it is possible in our framework to receive messages for specific tickets. As an added feature, message type filtering is supported at the task level. Any messages which fail to pass the filter are not delivered to the task, and are instead deallocated.

Another purpose of the transaction management system is task monitoring. We employ a publish/subscribe model for this purpose. Any task may request status information from another task by subscribing to its status information service, and then every status message published by that task will be sent to the subscribed task. At the moment, all status information is sent as unicast datagrams. The main purpose of this system is to notify the requester that its request has been received, and to notify it again when the request is completed. Other interesting applications of this technology could include distributed process schedulers that monitor the progress and system load on a cluster of nodes, and then schedule compute jobs to distribute the load to meet predefined criteria.

The resource management framework is tightly coupled with the index system. When a client program wants to access a resource, a query to the index system is made. The results returned can then be passed into the resource management object. The resource manager then attempts to open one of the resource from the result set. If possible, one resource from each canonical name in the result set will be opened. Thus, the resource manager is capable of overcoming node failures by looking for other copies of the same resource. The current implementation attempts to open one instance of every canonical name in parallel, and continues this iterative process as timeouts occur. Eventually, an instance of every canonical name will be opened, or the resource manager will run out of instances of a resource in the index result set.

The resource control system is built on top of a client-server framework. This framework was chosen because the types of resources we want to support are generally not concurrent objects. Thus, the resource management system consists of two REAP message types: a resource operation message, and a resource response message. Then, there are two types of resource objects: a client object, and a server object. For any given resource, there will exist exactly one server object, and one client object per task with an open handle to the resource. When a given client wants to perform an operation on the resource, it will send a resource operation message to the server object's transaction address. The server will then call the run method of the message, and through a set of polymorphic calls described below, it will perform I/O operations on the server object. A response message will then be sent to the originating node.

The client and server resource objects are based upon an abstract interface that defines several common methods that can be used on UNIX file descriptors. The major base operations are: open, close, read lock, write lock, unlock, read, write, and stat. In all cases, blocking and non-blocking versions of these functions are provided, and the blocking functions are simply built on top of the non-blocking code.

As a simple performance improvement, client and server caching objects were constructed that perform both data and metadata caching. Since our distributed resource interface is essentially identical to the virtual file system interface that unix-like kernels give to applications, standard locking semantics can apply. Thus, our caching module simply looks at the numbers open read mode and write mode file descriptors to determine the acceptable caching strategy. For the multiple readers, and single writer cases, we allow client-side caching. For all other cases we must disable client-side caching. Thus, our caching semantics are identical to those used in the Sprite Network Filesystem[6]. The REAP framework makes our implementation very simple because our mobile-agent based messages can easily turn on and off client caches with minimal overhead.

To demonstrate the power of this resource control model, we have built client and server objects to support a distributed shared memory architecture. Once again, we employ the abstract client-server caching model to increase performance.

The last major component of the REAP framework is process creation and management. This portion of the architecture consist almost entirely of message types. The primary message type is a process creation message. This message contains an index record pointing to the binary to execute. It also contains the argument and environment vectors to include, as well. A second message is process creation response message. This message simply contains the transaction address of the newly created process. Finally, task monitoring messages may be used to monitor the progress of a task using the publish/subscribe model discussed in the section on transaction management.

The REAP mobile code daemon permits us to experiment with many different mobile code paradigms over a fault-tolerant multi-platform framework. Because it provides a simple cross-platform, distributed interprocess communication framework, it is very useful for developing system of collaborating distributed processes. This approach is capable of mimicking all the major mobile code paradigms, as shown in [3]. Furthermore, its polymorphic code selection system permits us to use the optimal algorithm on a given system without significant user interaction. Finally, the distributed resource management system allows us to reduce bandwidth and permit concurrent use of resources without breaking normal concurrency rules.

5. SECURING INDIVIDUAL HOSTS

This section describes work done on the project related to protecting individual hosts. We explored the use of re-programmable hardware in security systems. A major byproduct of that research was the implementation of an FPGA based encryption engine with performance superior to all the other published implementations. We also showed how to combine compilers and hardware instruction sets to protect against covert channel attacks, like differential power analysis. Finally, we experimented with modifying intermediate representations of mobile code executables to prevent unauthorized execution and reverse engineering of software.

5.1 FPGA ENCRYPTION ENGINE

The recent evolution of powerful FPGA hardware has made their suitability for cryptoprocessor systems more evident. Additionally, most cryptographic algorithms have ease of hardware design as a main design goal, which makes them particularly well suited to implementation in Verilog HDL. The high cost of cell-based and full custom cryptography chips makes them prohibitive. Also, the inefficiency and low throughput of software implementations prevents their widespread use. FPGAs present an ideal compromise in that they retain the reconfigurability and control of software approaches while also achieving high throughputs near those of custom-designed ASICs.

All high-throughput cryptographic block cipher implementations have utilized a pipelined approach, where inner-round functions (such as those in AES or DES) are duplicated. This allows for both high throughput and efficient use of hardware. However, key control logic becomes complex should one desire to change keys during encryption, as either the pipeline must be emptied and the key changed or additional logic is required to detect and adapt to the change. We propose a high-throughput parallel processing an alternative to pipelined cryptoprocessor architectures. In addition to the pipelining benefits of hardware efficiency and high throughput, it allows for scalability and controllability of the resulting architecture. Pipelined FPGA cores do not utilize the entire chip; our parallel architecture allows for maximum utilization given sufficient I/O resources.

Conventional pipelined implementations of the AES standard can achieve data rates up to about 17.5 Gbps [Jarvinen 2003]. Pipelined implementations of DES can achieve data rates of up to about 10 Gbps, depending on both the target architecture and the design entry method [Jarvinen 2003]. By comparison, a high-speed software implementation of DES would likely achieve a throughput of about 250 Mbps. Our parallel architectures have indicated memoryless throughputs of 18.8 Gbps for AES, 9.00 Gbps for DES and 8.631 Gbps for 3DES. Using the Virtex-II Pro's Block RAM resources for the AES substitution boxes, we achieve a throughput of 17.7 Gbps. This paper aims to evaluate the performance and implementation details of parallel processing architectures based on the AES and DES symmetric key block ciphers. Verilog HDL modules are synthesized on the Virtex-II Pro FPGA platform to evaluate performance and security of parallel cryptoprocessing applications.

We propose the parallel architecture as a method of achieving maximum utilization of the FPGA's logic cells and I/O resources. Symmetric key block ciphers consume a very large amount of silicon area with respect to their I/O usage; for example, a memoryless 128-bit AES encryptor proposed in [Jarvinen 2003] uses roughly 12 times more area than an arbitrary 64-bit multiplier despite having the same amount of I/O usage. Typically, larger FPGAs are required for implementation of these block ciphers, and larger FPGAs have by definition higher numbers of I/O pins.

Table 1: Summary of Related FPGA Encryption Implementations

Design Origin	Implementation	Target Architecture	Throughput
Belfast, DSiP Labs	Pipelined DES	Virtex	3.87 Gbps
Xilinx	Pipelined DES	Virtex	10.7 Gbps
Sandia National Labs	Pipelined DES	ASIC	9.28 Gbps
Tampere University, DCS Lab	Pipelined 3DES	Virtex	364 Mbps
Rodriguez, Saqib, Diaz	Pipelined AES	Virtex-E	4.12 Gbps
GMU	Pipelined AES	Virtex	12.2 Gbps
Helsinki UT	Pipelined AES	Virtex-II	17.8 Gbps
University of Calgary	Multithreaded AES	Virtex-II	7.60 Gbps

As we will show, pipelined architectures require considerably more area than a single parallel encryption block; however, a fully parallel encryption architecture requires more area than a pipelined architecture. Parallel blocks allow a far greater degree of flexibility when designing an encryption system as we will detail based on its area flexibilities and security advantages. If two pipelined architectures cannot fit within a given device, an arbitrary amount of FPGA resources, both logic and I/O, will remain unused as the pipelined block cannot be split. However, individual parallel blocks are considerably smaller and can be used to reduce fragmentation and increase utilization of the FPGA's logic and I/O resources.

We will also show that parallel architectures provide both performance and utilization benefits in area-constrained devices. If the available area is an integer multiple of the area required for a pipelined architecture, pipelined systems have a performance advantage. However, for spaces larger or smaller than this, pipelined systems become inefficient. Note we make the assumption that additional I/O resources are always available. However, in a black-and-white area comparison, pipelined architectures are considerably smaller than fully parallel architectures. This is unavoidable, as the key hardware must be duplicated for each block. In the case of AES, the key scheduling module is rather large; this represents a direct tradeoff between area and security of the system.

It is important to both the security and functionality of the system that the keys are kept separate. This requires that each individual parallel block have its own key hardware, which enforces spatial isolation of the keys. This allows multiple independent encryptions to process simultaneously. As a consequence, the parallel encryption blocks suffer an area penalty with respect to the pipelined architecture. It should be noted that is infeasible to use shared key hardware among the parallel encryption blocks since each block is limited by design to only one output per cycle. Key sharing is impossible because even if two encryption blocks share a common key, no two parallel blocks are the same point in the encryption and hence would require separate key values.

Figure 34 below details the differences between our proposed parallel architecture and conventional pipelined architectures. Each block in the parallel architecture is a completely self-contained encryption unit. The dotted lines indicate the smallest possible unit that can encrypt a block of data. A fully parallel encryption architecture utilizes *n* blocks, where *n* is the number of rounds of the specified block cipher. Note that a pipelined implementation requires all *n* functional blocks whereas a parallel block requires only one. Thus we define a parallel encryption block as a single round function block and a key control module. Furthermore, we define a pipelined encryption as having one key control module and *n* round function blocks. In the parallel case, more than *n* blocks requires an additional I/O allocation. The parallel encryption blocks each have their own independent key hardware. This illustrates the property of *n* independent encryption sessions utilizing *n* independent keys. Also, it follows logically that only one independent encryption block must be present for encryption to proceed. We can see then that the use of *n* independent keys requires a minimum of *n* parallel blocks.

Figure 35 below illustrates the performance comparison of the fastest and most efficient published implementation of AES-128 with our fully parallel architecture. Note that this fully parallel architecture uses Block RAMs to implement the byte substitution boxes. We see that as expected, the pipelined implementation has better best-case performance but is limited greatly in terms of usability. It is clear that when area is constrained, parallel architectures provide improved performance and provide more efficient utilization of the FPGA's resources. The scalability of the parallel architecture makes it suitable for smaller spaces. The pipelined architecture requires approximately 11000 SLICEs and the parallel blocks each require approximately 1300 SLICEs. In this case, where the available area is a multiple of approximately 11000, the pipelined architecture has an advantage. However, for ranges larger and smaller than integer multiples of 11000, the parallel architecture provides greater logic utilization as well as increased overall system performance.

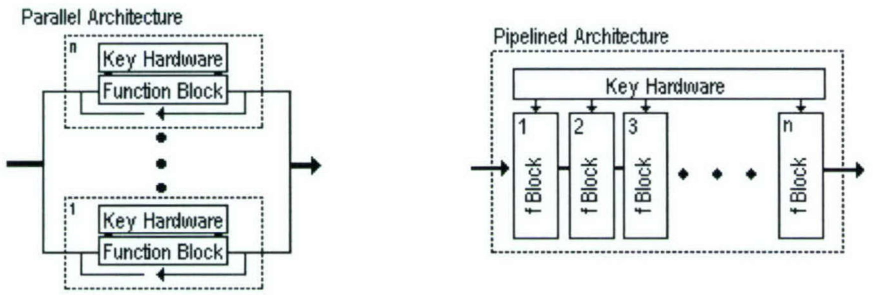


Figure 34. Overview of Parallel and Pipelined Architectures

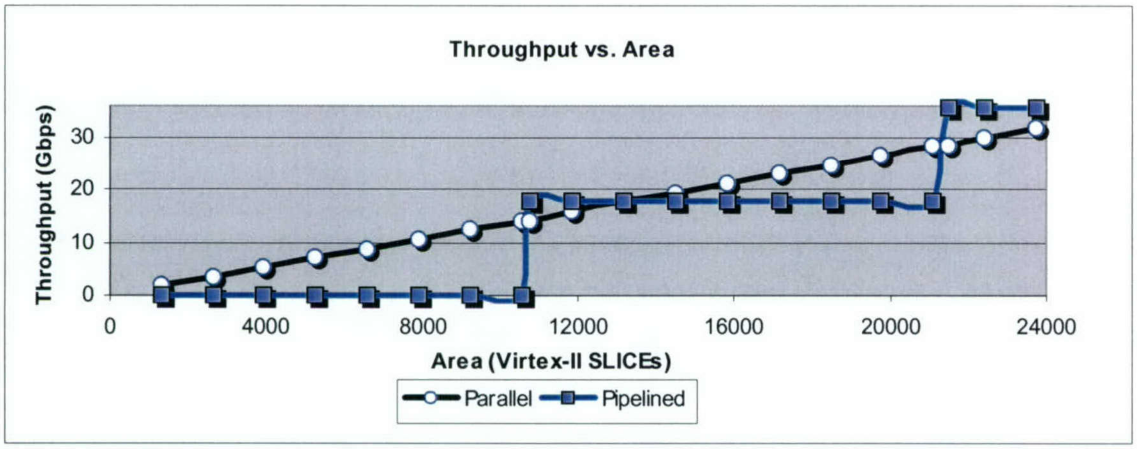


Figure 35. Area / Throughput Comparison of Parallel and Pipelined Architectures

The Advanced Encryption Standard (AES) was officially adopted in May of 2002 as the new encryption standard. It is designed to operate on all combinations of data input and keys with lengths of 128, 192, and 256 bits. This design uses a data input length of 128 bits with a key length of 128 bits. A block of data is placed into a 16-byte array, and proceeds through 10 rounds of encryption. Basic operations include byte substitutions, independent row byte shifts, column Galois field multiplications, and key additions. Row shifting and column multiplication use 32-bit operands (one 4 byte row or one 4 byte column). Similar to DES, this design is not pipelined; it is able to achieve reasonable throughput without doing so. Also, space limitations within the context of a mid-size FPGA make pipelining prohibitive. It should be noted that AES is not symmetric for encryption and decryption. The mathematical operations are different and require different hardware.

The design uses reusable function hardware, with minimal unnecessary hardware duplication. As was the case with DES, the structure of AES lends itself logically to reusable function block. The four row shifting operations are separate modules, since each operation is a separate shift. All row shifting is done through routing channels; no logic resources are used. They are similar to DES permutations, though entire bytes are shifted rather than individual bits. The column multiplication operations use four separate modules, allowing each column multiplication to proceed in parallel. The byte substitution is a 256x8 ROM lookup, and it is duplicated 16 times to also allow maximum parallelism. Also, we can use Virtex Block RAMs to implement the byte substitution tables. This saves a considerable amount of space, since a fully combinatorial implementation of a single encryption block requires 1,280 Virtex-II SLICEs for substitution tables alone. A dual-ported Block RAM is used to implement two substitution boxes. This requires 8 Block RAMs per block; a fully parallel block would use 80 Block RAMs. The substitution boxes associated with the key scheduler are implemented combinatorially for performance reasons. All other internal functions are combinatorial. This allows for a parallel architecture, which to some degree sacrifices hardware efficiency for throughput.

The security of AES has been well researched and is widely considered to be more secure than Triple DES. Its longer key length adds to its security capabilities; additionally, key lengths of up to 256 bits allow an even higher level of security. In this design, the 10-round structure and 128-bit block size allow the AES algorithm to encrypt data much faster than a similar Triple DES implementation. Also, AES provides more efficient use of hardware; its performance and security capabilities far offset its somewhat larger area.

The AES implementation is ultimately not as compact as possible, but duplicates some hardware to achieve higher performance. The byte substitution ROM is implemented 16 times; these modules are re-used each round during encryption and decryption. An extremely area-constrained design could theoretically use only one byte substitution ROM with a huge penalty to throughput. Also, the column multiplication function is repeated four times. This is not as much of an issue, since a single multiplication operation requires roughly half the area of a single byte substitution ROM. The largest component is the key scheduler, which is not duplicated. The bulk of the key scheduler is comprised of four byte substitution ROMs and four 32-bit XORs.

The encryption module is able to attain gigabit throughput, but as a comprehensive module the system must operate at or around the decryption frequency (depending on synthesis results). The overall throughput is reasonable at about 1.6 Gbps. Note that the decryption operation initially incurs a 10 cycle key setup penalty once per key lifetime. The keys are generated sequentially but must be used in reverse order. It should also be noted that encryption and decryption could occur in parallel with a comprehensive module provided the inputs arrive on subsequent cycles. Also, it is assumed that decryption key setup occurs prior to the bulk of the encryption or decryption operations. For this paper, we consider only encryption performance.

Since the design does not waste hardware, we can also construct a dedicated high-throughput AES encryption processor based on duplicated single AES encryption modules. Unconventional approaches have been proposed before, including multithreaded and pipelined approaches. This architecture is similar to a multithreaded approach in that synchronization between "threads" need only occur to prevent data collision at the output. Assuming all modules are identical, collisions are impossible, as a collision would require data arriving simultaneously to two separate units. This is prevented because the input bus is shared.

For a fully parallel encryption architecture, we do not include any decryption modules. Additionally, the inclusion of decryption would reduce the maximum hardware utilization to 50%. At best, it would interleave encryption and decryption operations, likely increasing the overall latency for both operations.

Note that we include synthesis results for both a single AES block and an AES encryption processor with a hard-coded key. This is similar to the JBits implementation of DES mentioned above, and the speed gains are noticeable. Also, a significant area reduction is achieved. However, this approach is impractical for two reasons. The key security is weakened greatly, as all round keys (including the key in its pure form) are stored in either on-chip RAM or ROM. Thus direct memory attacks could intercept the key itself. Also, changing the key requires a partial reconfiguration of the device. This expends a considerable amount of power.

Table 2 : Performance of AES Encryption and Decryption on Virtex-II Pro FPGAs

Algorithm	Number of Parallel Blocks	System Frequency	Area (Slices)	Key Units	Block RAMs	Throughput
AES-128	10	146.798 MHz	23979	10	0	18.80 Gbps
AES-128	10	138.122 MHz	14013	10	160	17.77 Gbps
AES-128	2 (1 Encryptor / 1 Decryptor)	124.906 MHz	6184	2	0	1.599 Gbps
AES-128	1	147.973 MHz	2921	1	0	1.894 Gbps
AES-128	1	145.052 MHz	1319	1	16	1.857 Gbps
AES-128	10	150.621 MHz	20249	0	0	19.28 Gbps
AES-128	1	161.577 MHz	2370	0	0	2.068 Gbps

We have shown that a parallel architecture for symmetric cipher encryption allows a higher degree of control over conventional pipelined architectures. Also, the parallel encryption architectures allow for multi-gigabit throughput for all symmetric ciphers. Single-chip performance of this parallel approach exceeds most commercially available pipelined cores. The proposed architecture uses parallel encryption blocks to achieve a high throughput zero latency design.

The implementation of the algorithms and encryption processors in Verilog HDL allow for efficient implementation in both FPGA and ASIC mediums. Also, unlike full-custom designs, optimizations and changes can be made quickly and easily. This allows for a high degree of scalability and controllability of the parallel architecture. Additionally, through slight design modifications we can show that the use of Block RAM for substitution boxes improves relative performance.

We have shown also that a parallel architecture provides a greater degree of security than conventional pipelined architectures. We can use controlled physical random functions to generate a device-independent hardware signature. With some slight algorithmic modifications, we can limit the existence of the key to partial transient values, and hence we protect the symmetric key from analysis and interception.

5.2 SECURE INSTRUCTION SET AND DIFFERENTIAL POWER ANALYSIS

Our energy masking approach is based on eliminating the input dependencies of an operation. Our approach is focused on four types of operations that are critical in the DES encryption: assignment operation, bit-by-bit addition modulo two (XOR) operation, shift operation and indexing operation. In our approach, we do not mask all the operations, but only the operations that use the secret key and those operations that use the data generated from prior secure operations. The compiler analyzes the code and identifies how these variables are used within the code. Then, for the operators that work on these variables, the compiler employs secure versions of the corresponding instructions. It should be emphasized that it is not sufficient to protect only the sensitive variables annotated by the programmer. This is because the variables whose values are determined based on the values of the protected variables can also be exploited to leak information. Consequently, such variables also need to be protected. We achieve this using a technique called forward slicing [Howitz 1990]. In forward slicing, given a set of variables and/or instructions (called *seeds*), the compiler determines all the variables/instructions whose values depend on the seeds. The complexity of this process is bounded by the number of edges of the control flow graph of the code being analyzed. After all the variables whose values are affected by the seeds are determined, the compiler uses secure instructions to protect them.

Data Initial Permutation (L0,R0) = PermuteIP(Data) **Key Permutation** (C0,D0) = PermuteK1(Key) *= denotes insecure assignment*	**Data Initial Permutation** (L0,R0) = PermuteIP(Data) **Key Permutation** (C0,D0) ← PermuteK1(Key) *← denotes secure assignment*

Mth Rounds
Left Side Operation
Lm = Rm-1
Mth Key Generation
Cm = Rotate(Cm-1,n)
Dm = Rotate(Dm-1,n)
Km =PermuteK2(Cm,Dm)
Right Side Operation
E(R)= PermuteE(Rm-1)
f(Rm-1,K) = S(E(R)(+) Km)
Rm = Lm-1 (+) f(Rm-1,K)
	Mth Rounds
	Left Side Operation
	Lm ← Rm-1
	Mth Key Generation
	Cm ← Rotate(Cm-1, n)
	Dm ← Rotate(Dm-1,n)
	Km ← PermuteK2(Cm,Dm)
	Right Side Operation
	E(R)← PermuteE(Rm-1)
	f(Rm-1,K) ← S(E(R) <+> Km)
	Rm ← Lm-1 <+> f(Rm-1,K)

Output Inverse Permutation Output=PermuteIP^{-1}(R16,L16)	**Output Inverse Permutation** Output=PermuteIP^{-1}(R16,L16)

(a) Original DES operations (b) Modified DES operations

Figure 36. Modified DES Algorithm.

Figure 36 shows how we modified the DES operations. Figure 36(a) shows the original DES operations. The first step is initial permutation of the plaintext. This operation does not use any secret key and hence does not require being secure.

The next operation is the key permutation. This operation obviously needs to be secure. Figure 36(b) shows how we modify this operation. In this figure, the symbol "=" corresponds to the original assignment (i.e., insecure assignment), and the symbol "←" indicates that the assignment is secure.

The next step contains the operations within each round. Since some of these operations require the secret key, and the operations are repeated in every round using the data generated from the previous round, we need to secure all operations inside this block. Note that the modified left side operation uses a secure assignment operation, although it does not operate on the secret key directly. This is because it uses the data generated from the previous round (for $\geq 2^{nd}$ round) that uses the secret key. In the right side operation, all the instructions need to be secure. Each round uses four types of secure operations: they are secure assignment, secure shift, secure bit-by-bit addition modulo two and secure indexing. Note that the S symbol in the figure represents the S-Box operation.

The last operation is the output inverse permutation. This operation does not need any secure instruction although it uses data generated from secure instructions as it reveals only the information already available from the output cipher. The following section explains how our secure instructions are implemented.

Our target 32-bit embedded processor has five-pipeline stages (fetch, decode, execute, memory access and write back) and implements the integer instructions from the Simplescalar instruction set architecture. Its ISA is representative of current embedded 32-bit RISC cores used in smart cards such as the ARM7-TDMI RISC core. We augment our target instruction set architecture with secure versions of select instructions. To support these secure operations, the hardware should be modified as explained below.

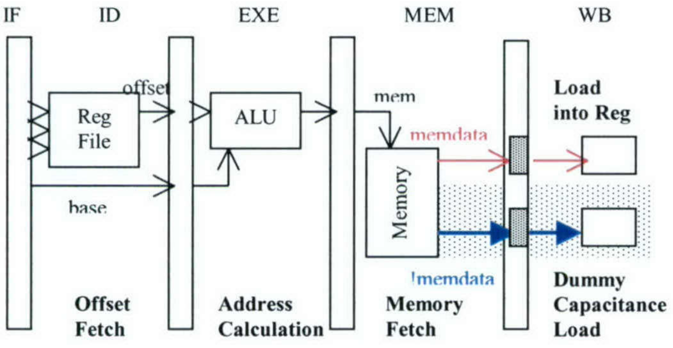


Figure 37. Secure Load Architecture (dotted portion is the augmented part)

First, we provide an overview of the underlying reasons for the differences in the power consumption because of data dependencies when executing these instructions. An assignment operation typically involves loading a variable and storing it into another variable. We will consider the parts of the load operation that are of interest. All stages of our pipeline (see Figure 37) till the memory access stage are independent of the loaded data (note that revealing the address of data is not considered as a problem). The memory access itself is not sensitive to the data being read due to the differential nature of the memory reads. However, the output data bus switching depends on the data being transmitted. For example, let us consider the different scenarios for the 1st bit (d0) of the 32-bit data read from the cache. If the values of d0 in two successive cycles are 0 and 1, it consumes more power than the case when the values are 0 and 0 in these two cycles. Specifically, for an internal wire of 1pF and a supply voltage of 2.5V, the first case consumes 6.25pJ more energy than the second case. The output from the memory access stage is fed to the pipeline register before being forwarded for storing the data in the register file. Thus, based on whether a bit value of one or zero is stored in the pipeline register bits, a different amount of energy is consumed. Finally, the energy consumed in writing to a register is independent of the data as the register file can be considered as another memory array.

The secure version of the load operation will need to mask all these energy differences due to bit dependences. This is achieved by the following modifications to the architecture. The buses carrying the data from a secure load are provided in both their normal and complementary forms. Thus, instead of a 32-bit bus, we use a 64-bit bus. Thus, the number of 1s and 0s transmitted in the bus will both be 32. However, this is not sufficient for masking the energy differences that depend on the number of transitions across the bus. But this modification along with a pre-charged bus can mask this difference. All the 64 bus lines are pre-charged to a value of one in the first phase of the clock. In the next evaluating phase, the bus settles to its actual value. Exactly, 32 of the bus lines will discharge to a value of zero. In subsequent cycles, energy is consumed only in pre-charging 32 lines independent of the input activity. The next modification involves propagating the normal and complementary values until the write back stage. The complementary values are terminated using a dummy capacitive load. The required enhancements to the underlying processor architecture are illustrated in Figure 3. Similarly, a secure version of the store operation involves passing along both the normal and complementary forms of the data read from the register file in the decode stage to the memory access stage.

A secure assignment uses a combination of both the secure load and the secure store to mask the energy behavior of the sensitive data. Figure 38 shows a specific elaboration of the use of the secure assignment in assembly code for the assignment performed during the "left side operation". The high-level assignment statement leads to a sequence of assembly instructions. The critical operations (the load and store instructions highlighted) whose energy behavior needs to be made data independent are then converted to secure versions in our implementation by the optimizing compiler.

```
.........
// Left Side Operation
for (i=0; i<32; i++)
        newL[i] = oldR[i]
.......
```

```
.................          .................
$L12:                      $L12:
  ...............            ...............
$L15:                      $L15:
  lw      $2,i               lw      $2,i
  ...............            ...............
  la      $4,newL            la      $4,newL
  addu    $3,$2,$4           addu    $3,$2,$4
  move    $2,$3              move    $2,$3
  lw      $3,i               lw      $3,i
  move    $4,$3              move    $4,$3
  sll     $3,$4,2            sll     $3,$4,2
  la      $4,oldR            la      $4,oldR
  addu    $3,$3,$4           addu    $3,$3,$4
  move    $4,$3              move    $4,$3
  lw      $3,0($4)           slw     $3,0($4)
  sw      $3,0($2)           ssw     $3,0($2)
$L14:                      $L14:
  lw      $3,i               lw      $3,i
  addu    $2,$3,1            addu    $2,$3,1
  move    $3,$2              move    $3,$2
  sw      $3,i               sw      $3,i
  j       $L12               j       $L12
$L13:                      $L13:
.................          .................
```

(a) Original Assembly Code **(b) Modified Assembly Code**

Figure 38. Code level representation of the left side operation

The secure 32-bit XOR instruction is implemented using complementary pre-charged circuit (see Figure 5) that will ensure that for every XOR bit that discharges in the required circuit, the complementary circuit will not discharge and vice-versa. In the first clock phase (when v =0), all (64 = 32 original + 32 complementary) the output nodes of the XOR circuit are pre-charged to one. In the next phase (when v=1), half of them will discharge and the other half of them will remain at one. In subsequent cycles that use the XOR, the energy is consumed only for charging 32 output nodes immaterial of the data activity.

During the S-Box operation, a 6-bit value is used to index a table. This operation is performed by a load operation with the 6-bit value serving as the offset in our underlying architecture. Note that our current secure load operation does not mask the energy difference due to differences in the offset. As these 6-bits are derived from the key, it is also important to hide the value of this offset. When the 6-bit value is added as an offset to the base address of the table, the addition operation will consume an energy based on the 6-bit value. In order to avoid this, we align the base address of the table such that the 6-bit value serves as the least significant bits of the lookup and the most significant bits are determined at compile time. Further, the inverted value of this 6-bit index is propagated to mask the energy consumption. Thus, the load operations used for indexing are replaced by the secure indexing that generates the memory address using our secure version.

In order to utilize these augmented architectural features, the compiler tags selected operations as secure. Secure instructions can be implemented using either the unassigned opcodes (bits in the instruction identifying the operation) in the processor architecture or by augmenting the original opcodes with an additional secure bit per operand. In our implementation, we resort to the second option to minimize the impact on the decoding logic. Whenever a secure version of the instruction is identified both the normal and complementary versions of the appropriate segments of the processor become active. For example, for the secure XOR operations, the data values (both source data and result data) are present in normal and complementary forms in the internal data buses. Further, the required and complementary versions of the circuit operate together. Since the additional parts consume extra power, the clock to the complementary versions is gated to reduce energy consumption. The details of the gating (note that the complementary version of the circuit is provided with a clock v gated with secure signal – secure v - for the evaluation phase) for the XOR unit implementation are shown in Figure 5. Thus, as opposed to energy consumption of 0.06pJ in the secure mode, the XOR unit consumes only 0.03pJ in the normal mode. Additional savings in energy also accrue during the execution of normal versions due to gating of the additional buses and the pipeline registers.

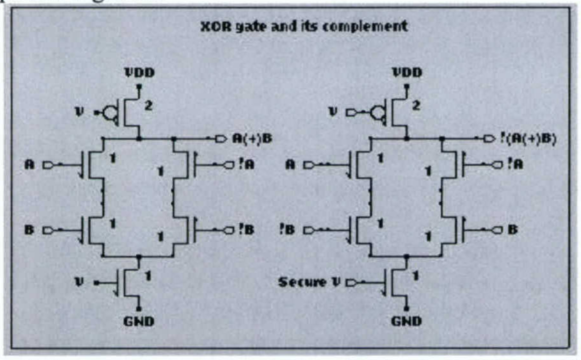


Figure 39. XOR circuit and its complement. v is the clock. A and B are the inputs to the XOR function

To evaluate the effectiveness of our approach, we have implemented the DES algorithm in software and captured the energy consumption in each cycle using a customized version of the publicly available SimplePower, a cycle-accurate energy simulator. We focus only on the processor and buses in this work, as memory power consumption is largely data-independent. The simulator uses validated transition-sensitive energy models for both the buses and functional units obtained through detailed circuit simulation, and is within 9% of actual values. It is able to accurately capture the differences in energy consumption due to data transitions. The flexibility of working with the simulator provides us the ability to monitor the energy consumed in every cycle (along with details of actual instructions executed) and also helps us in quickly identifying the benefits or (otherwise) in modifying the underlying processor architecture. Current measurement based approaches would be limited by the sampling speed of the measuring devices and would also be more difficult to correlate the operations and sources of energy consumption. The processor modeled for our simulation results is based on 0.25micron technology using 2.5V supply voltage.

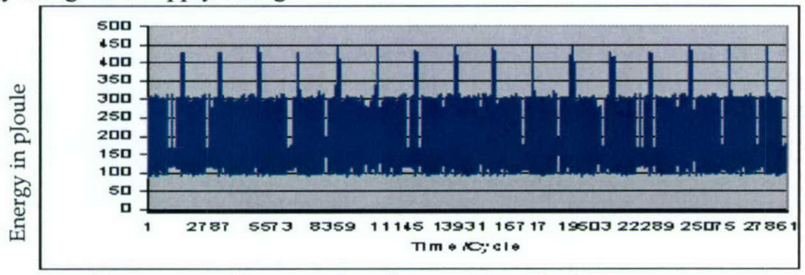


Figure 40. Energy consumption trace of encryption (every 10 cycles)

First, we show the energy behavior of the original DES algorithm to demonstrate the type of information that it leaks. Figure 40 shows the energy profile of the original encryption process revealing clearly the 16 rounds of operation. This result reiterates that the energy profile can show what operations are being performed. Next, we present a (differential) energy consumption trace for two different secret keys to demonstrate that the energy consumption profiles can reveal more specific information.

Figure 41 illustrates the difference in energy consumption profiles generated for two different secret keys using the same plaintext. This example illustrates that it is possible to identify differences in even a single bit of the secret key. Similar observations on energy differences can also be made using differences in one of key-related variables generated internally.

Figures 42 and 43 show the difference between the two energy consumption traces generated using two different secret keys and the same plaintext before and after the energy masking. These traces are shown only for the first round of DES algorithm for clarity. The graphs clearly demonstrate that using secure instructions can mask the energy behavior of the key related operations. While the effectiveness of the algorithm is shown using differences between profiles generated from two different keys, the results hold good for other key choices as well. Specifically, the mean of the energy consumption traces which generate different internal (key related) bits will not exhibit any differences that can be exploited by DPA attacks.

Figures 44 and 45 depict the difference between the energy consumption traces generated using two different plain texts but the same secret keys.

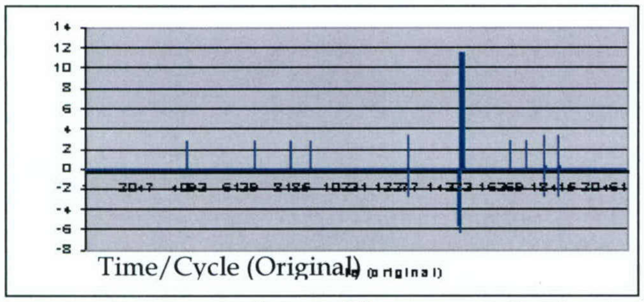


Figure 41. Difference between energy consumption profiles generated using two different secret keys (vary in bit 10), 1st round

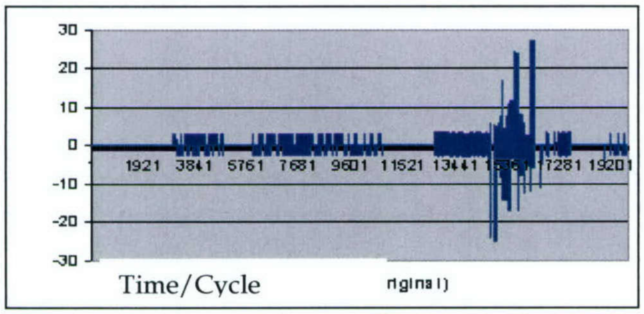


Figure 42. Difference between energy consumption profiles generated using two different keys before masking process

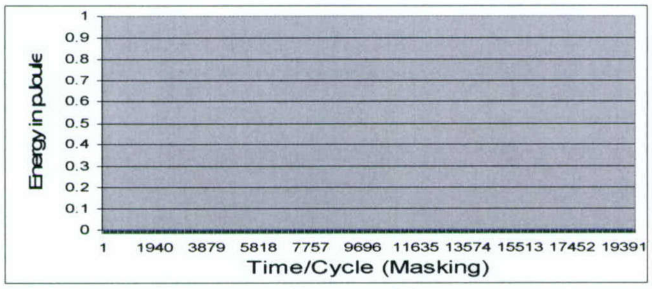


Figure 43. Difference between energy consumption profiles generated using two different keys after masking process

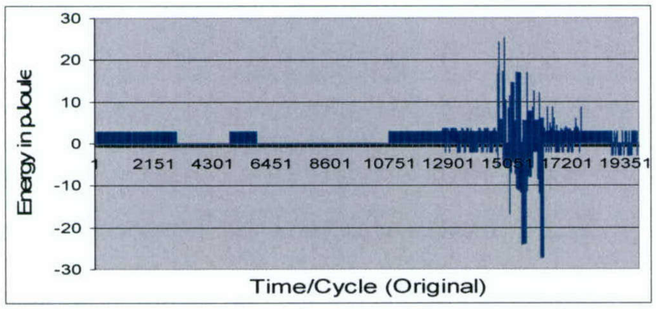


Figure 44. Difference between energy consumption profiles generated using two different plaintexts before masking process

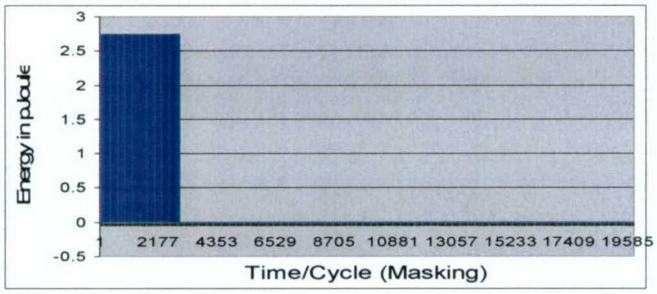


Figure 45. Difference between energy consumption generated using two different plaintexts after masking process

The first operation in the DES is plaintext permutation. Since this process is not operated in a secure mode, the differences in the input values result in the difference in both the energy masked and original versions. The other operations in the first round are secure; as a result, there are energy consumption power differences.

However, the proposed solution is not without its drawbacks. The energy masking requires that the same amount of energy be consumed independent of the data. Thus, additional energy is consumed in the circuits added for the complementary portion of the circuit as shown in Figure 46 However, this additional energy is 45 pJ per cycle (as compared to an average energy consumption of 165 pJ per cycle in the original application). Note that we add excessive energy even in places where the differential profile in Figure 8 shows no difference.

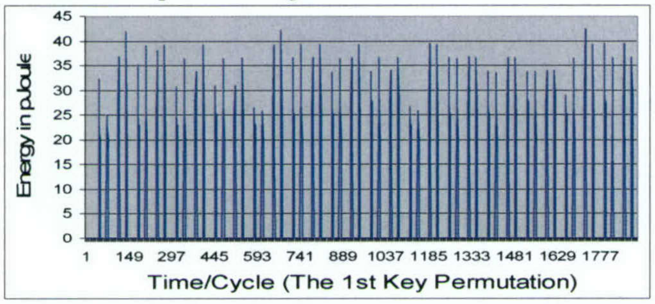


Figure 46. Additional energy consumed due to the energy masking operation during the 1st key permutation

This is because the same secure instruction is used for parts of the input that are the same for both the runs. Of course, in portions where the data was identical we have nothing to mask but we need to be conservative to account for all possible inputs in a statistical test using large samples. It must also be observed that our approach of using selective secure instructions helps to reduce the energy cost as compared to a naïve implementation that balances energy consumption of all operations. For example, looking at code segment shown earlier in Figure 4, we increase the energy cost of only one of the four load operations executed in the segment. On the other hand, the naïve approach would convert all the four load operations into secure loads thereby consuming significantly more energy than our strategy.

The total energy consumed without any masking operation is 46.4 uJoule. Our algorithm consumes 52.6 uJoule while the naïve approach consumes 63.6 uJoule (all loads and stores are secure instructions). When all instructions are secure instructions, it will consume almost as twice as much as the original, 83.5 uJoule. This scheme is the one used in current dual-rail solutions.

Smart cards, unlike magnetic stripe cards, can carry all necessary functions and information on the card. Therefore, recent years have witnessed a significant increase in smart card use throughout the world. In fact, Data Monitor predicts that over 3 billion smart cards are in circulation worldwide. As a result, ensuring secure use of smart cards is receiving a lot of attention.

The uniqueness of our solution comes from the fact that, unlike many previous techniques, we approach the problem from an architectural perspective and consider adding secure instructions to a given architecture. The purpose of these secure instructions is to hide the energy behavior of sensitive variables in the application (e.g., key values). Our experiments with the DES application demonstrate that the proposed solution is very effective in preventing the information leakage due to power analysis.

5.3 SECURE CODE DELIVERY

The proliferation of constrained embedded devices [requires a complete rethinking in the design of software. In contrast to desktop and server environments, embedded applications need to consider the stringent limitations imposed on several resources such as memory size and energy budget. The focus of this work is on providing a resource-conscious solution for supporting secure programming of remote devices.

In many embedded devices, the functionality needs to be reprogrammed periodically to support software upgrades or reprogramming may be required to adapt the functionality to changing operational needs. For example, the functionality of sensor nodes may need to be changed based on newly sensed events and the constraints imposed by limited memory space will necessitate remotely reprogramming the device as opposed to storing all envisioned codes locally. However, the flexibility of field upgrades provided by such remote reprogramming also makes the code vulnerable to eavesdropping and execution in an unauthorized device. In embedded environments such as sensor networks deployed in military applications or for natural disaster management, ensuring code protection becomes vital. There are two aspects of security that need to be addressed: authentication and privacy. Authentication is the verification of the source of information (in our context both programs and data are information). Privacy is restricting access to information to authorized entities.

Cryptography is the accepted tool for achieving these goals. A cryptosystem has a plain text space, a cipher text space, and a key space. Functions map data between the plain and cipher text spaces using the key. If both mappings use the same key, the process is symmetric cryptography. If the keys differ, the process is public key cryptography. Cryptosystems use mapping functions, whose solution without the key is of provably high computational complexity. Unfortunately, for many constrained embedded systems, cryptography is expensive in terms of time, power, and computational resources.

To overcome the resource needs of cryptosystems, a multi-tiered approach is sometimes used. Computationally expensive public key methods are used to exchange symmetric key at infrequent intervals. The less expensive symmetric key cryptography is used to frequently exchange symmetric hash functions. Data is exchanged after one execution of the hash function. On reception, the hashed data is sent through the hash function again; restoring the data to its original state. In this case, hashing is a form of obfuscation. Obfuscation serves a role similar to cryptography, but is less computationally intensive and has no computational complexity guarantees.

In this work, we propose an obfuscation-based approach for providing security of programming a remote embedded Java device. Our choice of Java is motivated by the following reasons. First, in the embedded domain, a variety of system and operating configurations are prevalent making it an attractive option to use architecturally neutral Java bytecodes. Second, Java technology is being increasingly supported in many embedded devices ranging from smart cards to cell phones. In our design, the goal is to permit mobile devices to be reprogrammed by transmitting the bytecodes associated with the new functionality while preventing unauthorized mobile devices from correctly executing these bytecodes.

In our approach, the programming system and the authorized embedded node exchange a Java bytecode substitution table using standard encryption techniques. Two protocols are given for bytecode table exchange: one uses symmetric and the other public keys. The substitution table is used by the user to encode the bytecodes to be transmitted and by the authorized mobile node to interpret the received bytecodes. Unauthorized bytecode installations will either cause an error during the verification process since lack of substitutions will most probably lead to stack overflows/underflows or prevent the intended operation from executing correctly. However, similar to all substitution ciphers, our scheme is vulnerable to frequency attacks by an eavesdropper. Hence, we design our substitution table such that the frequency information is minimized.

We validated our approach using a set of Java applications and show that the proposed technique is a performance effective solution as compared to a standard encryption technique based on Rinjdaels algorithm. Furthermore, our experiments show the robustness of the substitution-based approach to frequency attacks using the entropy metric.

Our approach complements prior solutions that attempt to protect mobile codes. Among approaches used for providing security for Java bytecodes include proof-carrying codes, modified class names, the use of encryption and filters. Shin et al. [Chander 2001] modify the bytecodes such that some classes and methods of interest are changed to more restrictive classes and methods. For instance, to prevent a window consuming attack, they modify the frame class to safe$frame class that can prevent and control every window generation. Necula et al [Necula 1996] introduce proof-carrying code that is based on the idea of transmitting the safety proof along with the application code. One of the problems with this approach is the large increase in code size due to the proofs.

There are also several suggested methods for protecting Java class files from illegal reverse engineering using de-compilation. One such method is to encrypt and decrypt the code at the server and client end respectively to limit access to authorized devices. However, encryption/decryption is expensive, especially in resource-constrained environments. Another method is to require end-users make code requests to download a class file to also provide the distributing server information on the platform the program will be executed on. The server would then compile and send the native code to the client instead of the bytecode form. This scheme may deter the reverse engineer by making the de-compilation process more involved. However, there are de-compilers for native code as well such as Valkyrie for Clipper, ReFox for FoxPro and dcc for C. Furthermore, the transmission of code in native form is unattractive when a larger variety of platforms need to be supported. Code obfuscation is an attractive alternative to make reverse engineering using decompilation difficult. In this technique, the class files are transformed to an obfuscated class file that produces the same output but is more to de-compile. In contrast to data-flow or control-flow obfuscations that make it more resilient to decompilers, our approach to obfuscation changes the opcode assignments associated with the bytecodes.

In our method, we use a bytecode conversion table (BCT) based approach instead of encrypting the original file that contains the application. The BCT contains a list of bytecode pairs <Op1, Op2>, where Op1 is the original bytecode and Op2 is the corresponding substitution. This conversion table is used at the authorized sender side to transform the original code into an encrypted code that is subsequently sent to the mobile devices.

While using substitution to obfuscate information may seem an easy solution, the main challenge is in thwarting attacks based on frequency analysis. Frequency attacks are common to all substitution ciphers. By counting the frequency of symbols in the cipher text and exploiting homeomorphisms in the plain text, substitution ciphers can be broken. Symbol frequency and homeomorphisms describe structure in the plain text domain that substitution ciphers carry over into the cipher text domain. In our case, it is possible for an eavesdropper to utilize the frequency of bytecode usage in applications as a means to deciphering the substitution text. For example, the usage of stack manipulating bytecodes tends to exhibit a higher frequency across different applications. We show how to use the metric of entropy in evaluating the resilience of the chosen substitution to frequency attacks and investigate different alternatives for substitution using this metric.

The confidentiality of a class file is achieved by using the BCT, assuming that the attacker does not have access to the BCT. This table acts as an encryption algorithm. Thus, a critical aspect of our system is to securely transmit this table between the programming system and the authorized device. In order to do that, we use a cryptography algorithm. Although cryptography itself is a costly process, we only need it when we change the contents of the BCT. This is less computationally intensive as compared to using encryption for the entire application code. Next, we show two different approaches in establishing this BCT securely.

A simple public key protocol for BCT exchanges is given in Figure 47. In the following discussion, we describe the protocol using the following terminology: WR- User Workstation private key; WU- User Workstation public key; IR- Mobile device private key and IU-Mobile device public key. The new BCT is established on the user workstation W.

To put the table into use, W sends the authorized device I a packet containing the following data structure:

```
Encrypted with IU{
        1.1 New byte table
        1.2 time stamp
        1.3 sequence #
        1.4 Encrypted with WR{Hash of 1.1-1.3}
}
```

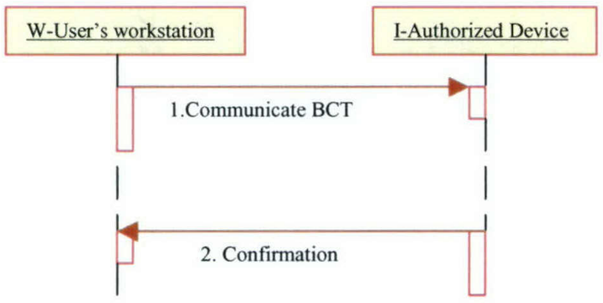


Figure 47. Public key protocol for installing the BCT on the mobile device

This packet contains a time stamp, a sequence number and a hash value encrypted using the private key of the user workstation. The role of the time stamp and sequence number is to prevent replay attacks by other devices. While the encrypted hash value helps the mobile device authenticate that this packet originates from the user workstation. When I receives the packet, it decrypts the contents using its private key IR. A hash value of the decrypted contents is computed and compared with the value in 1.4 decrypted using the workstation public key WU. If they are equal, this verifies that the message originated from W and has not been tampered with. In response, I sends the following message to W:
Encrypted with WU{
1. Sequence # +1
2. Encrypted with IR{ Hash of message 1}
}
W uses this message to verify that I received the message in an unaltered form and installs the BCT. A symmetric key variant of the can be found in [Saputra 2004]. Both protocols have the BCT update process initiated by the workstation. If it should be desirable for the mobile device to formulate the BCT, the roles of the two entities could be reversed. The sequence number could also be replaced with a random nonce. This would not significantly affect the protocol's security.

One-to-one byte code mapping is a mapping that substitutes one bytecode with another. An example of one-to-one mapping is a mapping from 0x4b (astore_0) to 0x2a (aload_0). This one-to-one mapping does not change the size of the original class file nor does it hide the instruction frequency profile. However, these substitutions can cause verification problems. One such example is the mapping of bytecodes that load operands onto the stack to unused bytecodes in the virtual machine. Hence, a code reaching an unauthorized mobile device will encounter a stack underflow during verification when instructions that consume the stack data are encountered.

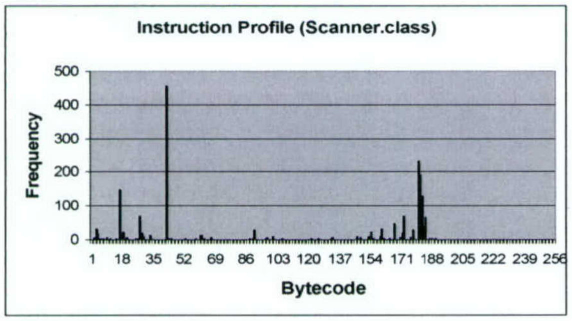


Figure 48 Original Bytecode Frequency Profile - Scanner.class

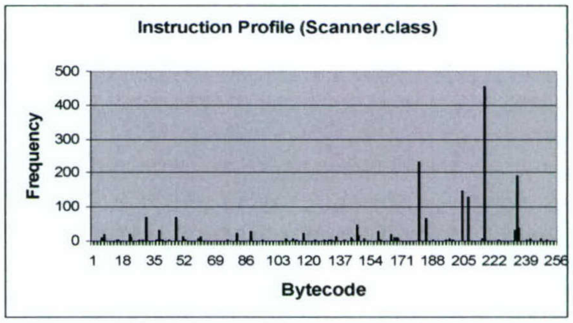


Figure 49. Bytecode Frequency Profile after (1-1) mapping - Scanner.class

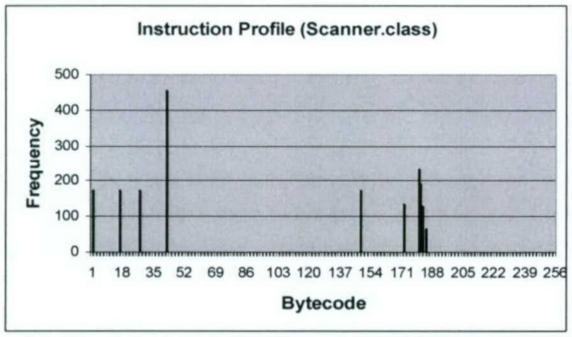


Fig. 50. Bytecode Frequency Profile after (M-1) mapping - Scanner.class

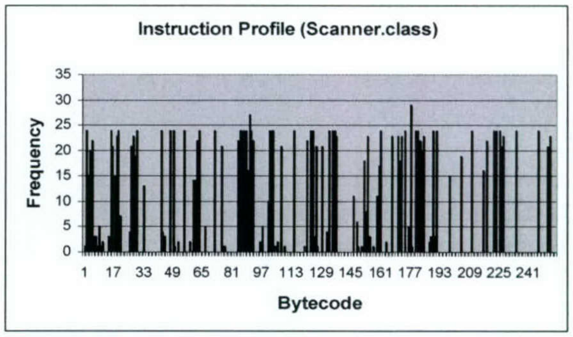


Fig. 51. Bytecode Frequency Profile after (1-M) mapping - Scanner.class

Figure 49 shows the impact of (1-1) mapping on the original frequency profile shown in Figure 48 for one of the representative classes (Scanner). It can be observed that the frequency profile is changed to a permutation of the original profile. This provides some additional protection if bytecodes of similar frequencies are interchanged. However, it is still quite vulnerable to frequency-based attacks.

Next, we consider a many-to-one byte code mapping approach. This mapping tries to combine bytecodes that occur infrequently into one single bytecode. To support the many-to-one mapping, we introduce the concept of extended bytecode. The extended bytecode uses two bytes to represent the opcode. The first byte is the opcode of the new extended instruction, and the following byte is identifies the original bytecode. The purpose of this many-to-one mapping is to combine the low frequency bytecodes into a single bytecode with a higher frequency. While this transformation skews the frequency profile as observed in Figure 50, it still does not hide the information on the frequently occurring bytecodes. However, as will be shown later, this technique is a useful pre-mapping step when combined with other approaches. In addition, the size of the resulting substituted class file will be larger due to the additional byte in each extended bytecode. Consequently, the field associated with the code-size in the Java class file is also updated when employing this substitution.

The next option is one-to-many mappings. Since the frequency profile of different Java bytecodes is not uniform; there are some bytecodes that occur more frequently than the others. This property can be exploited by frequency-based attacks. In order to make such an attack difficult, different occurrences of a frequently used bytecode can be assigned to different unused bytecodes. For instance, the frequently occurring a_load0 bytecode can have three different opcodes associated with it. The ability to reduce the frequency of these frequently occurring bytecodes is limited by the availability of unused (within the JVM) bytecodes. It must be observed that the use of M-1 mapping can increase the availability of unused bytecodes and increases the effectiveness for the 1-M mapping.

Figure 51 shows the impact of (1-M) mapping on the original frequency profile shown in Figure 4. It can be observed that the frequency profile becomes more uniform, thereby decreasing the information content that can be used by eavesdroppers.

While the mapping schemes have been discussed above individually, the resilience to frequency-based attacks can be further increased by using these mappings in a combined fashion. The combinations considered in this work are:

- One-to-many, Many-to-one, and One-to-one Combination - OM: This scheme first employs (1-M) mapping followed by (M-1) mapping and then by (1-1) mapping. It should be noted that this whole mapping can be done during a single pass to encode the transmitted bytecode seqeuence. (1-M) and (M-1) mappings are used to flatten the instruction profile while the (1-1) mapping is used to shuffle the profile.
- Many-to-one, One-to-many, and One-to-one Combination - MO: This scheme first employs (M-1) mapping followed by (1-M) mapping and finally by (1-1) mapping.

Note that OM and MO differ in that each applies the mappings in a different order.

In order to compare the quality of the obfuscation provided by mappings described above, we borrow concepts from information theory. To measure the quality of data coding, Shannon employed the concept of entropy to measure the presence of structure in data streams [1]. Structure in data streams implies predictability. Predictable events provide less information than unexpected events. This logic supports using the frequency of symbols in a text as a measure of their information content. For example, a symbol (in our case bytecode) that occurs 50% of the time provides less information than one occurring 10% of the time. Using pa to represent the frequency of symbol a (where a ÎS, and S is the alphabet used) as a fraction of the data stream, the entropy of the data stream is defined as $-\sum_a p_a \log p_a$, where log is the base two logarithm. Since pa is constrained to values between zero and one, the logarithm is always non-positive and all elements of the summation are non-negative. It is easily verified that this function has a maximum when all symbols are equally likely, i.e. occur with frequency $1/|\Sigma|$. This occurs when a histogram of the occurrences of symbols in a representative set of strings is flat. The entropy (information content) is maximized at that point and the amount of structure is minimized. These insights are well established and the basis of data compression techniques. In this work, we use entropy as a measure of resilience to frequency-based attacks. Lack of structure in the data streams makes it more difficult to decipher the information content of the data streams.

To evaluate the effectiveness of our mapping schemes, we used 52 different classes from various Java application suites: SPEC JVM98, Volano mark, UCSD, and DigSim. Our methodology for choosing the appropriate substitutions focused on trying to flatten the histogram associated with the frequency of bytecode usage. We utilized two parameters: the mean of the frequencies of all bytecodes and a user-specified threshold, T, that indicates the amount of tolerance to variations below and above the mean in the flattened profile. Thus, bytecodes that have their frequencies in the range of Mean - T and Mean + T are considered to be already balanced (see Figure 52). The bytecodes that have frequencies greater than Mean + T are considered for (1,M) transformations and M is set to the frequency of that bytecode divided by the mean. This process is repeated for all bytecodes with frequencies greater than Mean + T as long as there are available unused bytecodes. A similar process is used to merge bytecodes with frequencies less than Mean - T using the (M,1) mapping. When selecting (1-1) mappings, the bytecodes that have the minimum difference in their frequencies are selected for swapping. The results from these steps are then used in creating the BCT entries that capture the corresponding substitutions.

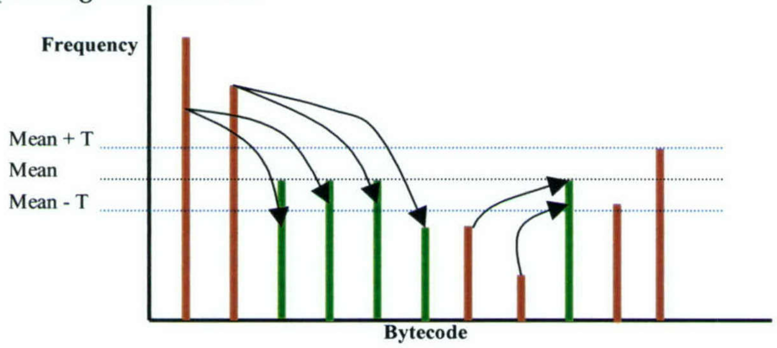


Figure 52. Flattening the profile

It must be observed that this process of creating the BCT can be performed either using the profile of an individual Java class or using the profile of a set of Java classes. While the use of a custom BCT for a class will help flatten the profile better, it is more practical to create the BCT using the entire class of applications. The latter approach helps to limit the BCT setup cost that can be amortized across several class file transmissions. In all our experiments, we use the profile obtained across all class files (from different applications) in creating our BCT.

In order to assess the resilience of our different mapping schemes to frequency attacks, we evaluated the entropy values of the resulting substituted Java class files. While we performed our evaluation for 52 different class files, we only show results of representative class files picked from the different benchmark suites in the Table. The last row in this table corresponds to the results averaged across all the 52 class files (not just those shown in Table 1). We do not include a separate column for the (1-1) mapping as it has the same entropy as that of the original one. As mentioned earlier, a higher entropy value indicates more resilience to frequency based attacks. It can be observed that all combinations except the (M-1,1-1) combination increase the entropy, thereby reducing the information content available for attackers using frequency-based profiles. An interesting observation is that using M-1 before applying the 1-M transformation results in the highest entropy values. Consequently, the MO scheme produces the most robust substitutions among the different schemes explored. This is because the M-1 mapping frees additional slots of bytecodes that can subsequently be exploited by the 1-M mapping.

Table 3. Entropy values resulting from different mapping schemes

	Entropy				
Class file	**Original**	**1-M, 1-1**	**OM**	**M-1,1-1**	**MO**
ParameterFrame	4.17	6.08	6.02	4.14	6.55
Parser	4.49	6.41	6.22	4.35	6.76
Plasma	4.75	6.37	6.16	4.65	6.86
Probe	4.62	6.37	6.32	4.62	6.99
QubbleSort	3.86	4.79	4.73	3.79	5.21
RuntimeConstants	3.01	5.5	5.5	3.01	5.95
Scanner	4.29	6.22	5.96	4.16	6.63
SchematicPanel	3.71	6.11	6.03	3.67	6.71
SelectionSort	4.18	4.72	4.63	4.05	5
ShakerSort	3.97	4.64	4.54	3.88	5.17
ShellSort	4.36	4.9	4.74	4.24	5.29
P	3.42	6.41	6.4	3.42	6.92
Q	4.65	6.37	6.28	4.63	6.75
Vmark2_1_2_0	4.95	6.49	6.27	4.82	6.84
Averages	4.12	5.64	5.55	4.06	6.07

In addition to preventing frequency attacks, we observed that the substituted class files failed the verification process in all the 52 classes tested using all the mapping schemes. The verification failed for a variety of reasons such as illegal target of jump or branch, illegal location variable number, illegal instruction found at offset and unable to pop operand off an empty stack location.

Table 4. Entropies of different mapping schemes measured by tracking all the classfiles

Entropy of 1-M, M-1, and 1-1				**Entropy of M-1, 1-M, and 1-1**			
Original	1-1	1-M, 1-1	OM	Original	1-1	M-1, 1-1	MO
5.19	5.19	7.38	7.17	5.19	5.19	5.08	7.95

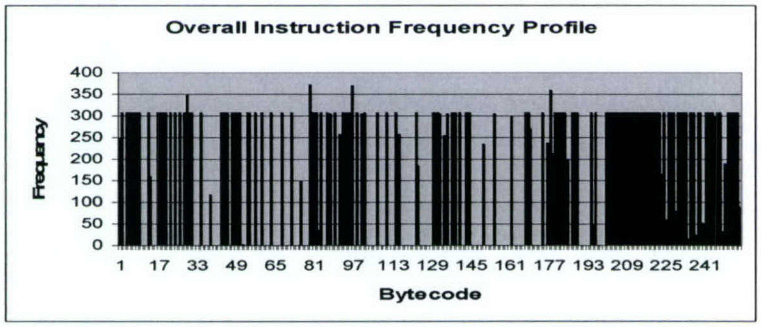


Figure 53. Overall Bytecode Frequency Profile Using OM

Since the attacker could also use cumulative information across all transmissions, we now show entropy values using the resulting frequency profile from all applications (See Table). Given that our methodology is based on the global mean using all profiles, it is evident that these entropy values are higher than those of individual applications that may have a profile behavior different from the general trend. We also include the histograms of the profiles resulting from using OM and MO in Figures 53 and 54. It is evident looking at these figures that the higher entropy metric of 7.95 for MO translates to a better flattening of the histogram as compared to the 7.17 entropy value for OM.

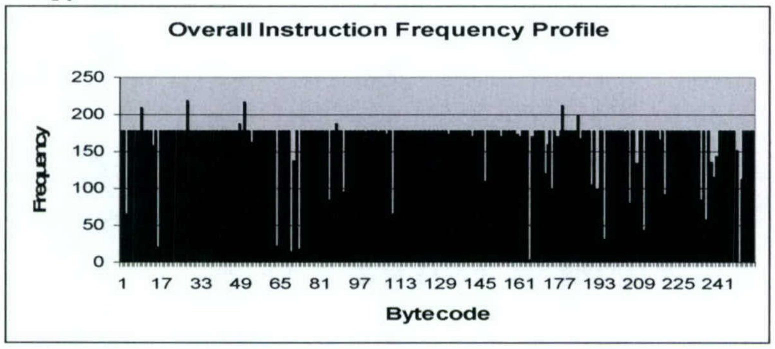


Figure 54. Overall Bytecode Frequency Profile Using MO

Shifting focus from evaluating the proposed mappings based on security metrics, we now evaluate their impact on performance. Specifically, we measure the time required for performing the substitutions at the sender and the reverse substitutions at the receiver and contrast it with a commonly used encryption technique, Rijndael[13]. When using the encryption algorithm, we perform encryption at the sender and decryption at the receiver. The table shows the results obtained using the MO mapping that measured best on the entropy metric and compares it with results from Rinjdaels algorithm. Due to the less computationally intensive nature of the proposed mapping, MO is about five times faster than the encryption approach on the average when executed on the same SPARC workstation. It must be observed that there would also be corresponding savings in energy consumption due to reduction in computational complexity. These savings are particularly important for resource-constrained environments. It must be noted that we do not include the time for the BCT setup in the MO mapping as it is negligible when amortized over sending different codes.

Table 5. Performance Comparations between the MO mapping and Rijndael

Class File	T_substitution (uSecond)	T_desubstitution (uSecond)	T_encrypt (uSecond)	T_decrypt (uSecond)
ParameterFrame	306	264	1079	1060
Parser	2309	1946	6425	6348
Plasma	417	324	1267	1284
Probe	262	219	973	986
Qsort	157	133	494	509
RuntimeConstants	1026	874	4704	4645
Scanner	1620	1361	3302	3232
SchematicPanel	1250	1053	3547	3513
SelectionSort	124	108	436	426
ShakerSort	155	130	482	475
ShellSort	140	119	466	459
P	7146	5185	6449	6644
Q	2626	1925	3096	3071

Table 6. Size Comparisons between original class files and substituted class files

Class File	Size_original (bytes)	Size_substituted (bytes)	Differences (+bytes)
ParameterFrame	2983	3057	74
Parser	24146	24938	792
Plasma	3737	3868	131

Probe	2605	2649	44
QSort	724	755	31
RuntimeConstants	17852	17858	6
Scanner	11730	12168	438
SchematicPanel	12814	13117	303
SelectionSort	547	567	20
ShakerSort	684	723	39
ShellSort	583	613	30
P	24689	24931	242
Q	11397	12691	1294

We also analyzed the impact of the MO mapping on the resulting size of the substituted class files and present these results in Table 4. We observe that on an average the size increase is less than 1.7%.

In addition to using the frequency of individual bytecodes, a frequency-based attack could also exploit the high occurrences of particular sequences of bytecode instructions. For example, a sequence such as (iload_0, iload_1) is commonly used due to the stack based nature of the Java virtual machine. To illustrate the impact of our optimizations on sequences, we find the frequency of two sequence instructions before and after applying the MO mapping strategy. Figure 11 shows the original frequency profile and the X axis represents the sequences starting from [0x0] [0x0 - 0xFF] followed by [0x1][0x0 - 0xFF] until [0xFF] [0x0 - 0xFF]. Here, for example, the sequence [0x1b] [0xb5] represents the putfield bytecode (opcode = 1b) followed by the iload_1 bytecode (opcode=b5).

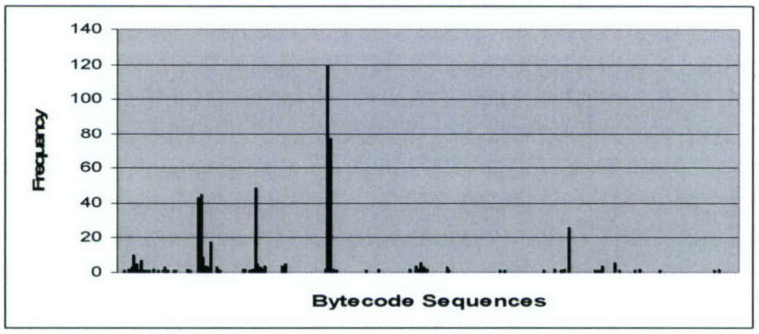


Figure 55. Original Frequency Profile of Bytecode Sequences for Scanner.class

Figure 56 shows how the frequency profile changes from Figure 11 when we apply the MO mapping. We observe that the profile is obfuscated due to the mapping. As a specific example, the highest frequency in the new profile is obtained for the bytecode sequence (putfield, iload_1) as opposed to (aload_0, aload_1) sequence in the original profile. However, as the profile still reveals frequency variations, we also explored an approach based on instruction folding.

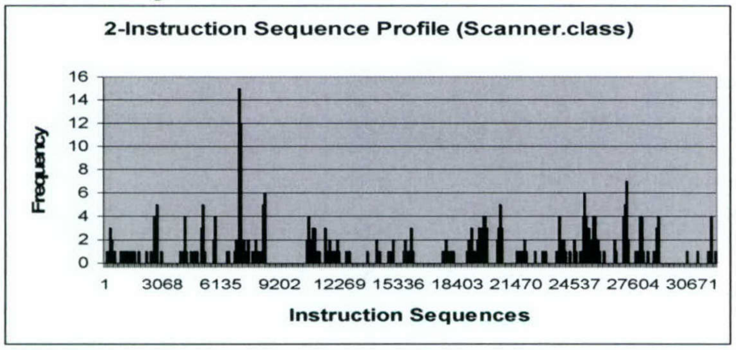


Figure 56. Frequency Profile of Bytecode Sequences for Scanner.class using MO

Instruction folding is a method that is used by bytecode processors to combine commonly occurring sequences into one single operation. Sequence mapping takes advantage of this approach in conjunction with other techniques discussed earlier to reduce the frequency information of bytecode sequences. Figure 57 shows the influence of applying sequence mapping for the pair (putfield, iload_1) on the profile given in Figure 56. While not shown here due to lack of space, sequence mapping also helps to reduce the size of the class file and also affects the frequency profile of individual bytecodes. In general, sequence mapping is an effective mechanism to hide the frequency information associated with larger sequences of instructions.

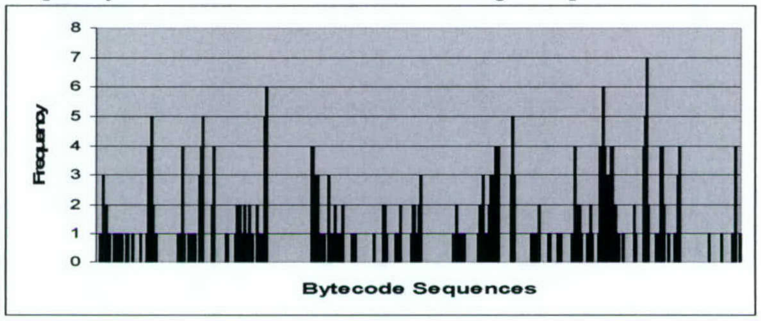


Figure 57. Impact of sequence mapping on profile shown in Figure 56

Cryptanalysis is used to determine the security of encryption techniques. Attacks are divided into classes based on the types of information available to the attacker, they are:

- Chosen Plaintext based attack. In this attack, the attacker knows some set of plaintext - ciphertext pairs for plaintexts of the attacker's choice. An example of a chosen plaintext based attack is differential cryptanalysis. Our approach can be vulnerable to this kind of attack where an opponent can choose a set of plaintexts that have specific differences and the associated encoding. Inferring the mapping in this case would be trivial. However, to obtain the associated encoding, the attacker needs access to either our translation table or know the plain text being transmitted. Due to the use of standard encryption for transmitting the translation table, the access to translation table is restricted. The second vulnerability is mainly an issue only when the codes being transmitted are limited to a known few codes.
- Known Plaintext based attack. The attacker knows some plaintext-ciphertext pairs. The difference between known plaintext and chosen plaintext based attacks is the set of the plaintext-ciphertext pairs that the former has, is out of the attacker's control. On the other hand, in the chosen plaintext based attack, the pairs are based on the plaintext that the attacker chooses. One example of known plaintext based attack is linear cryptanalysis . The vulnerabilities to this attack are similar to those discussed for the chosen plain text approach in the worst case.

The more appropriate attacks of concern for our application are ones where the attacker has access to a set of encrypted files, such as Algorithm-and-Cipher based attack and Ciphertext-only based attack.

- Algorithm-and-Ciphertext based attack. Attackers have the algorithm and a set of ciphertext they do not have access to the plaintext. In this situation, they try to use a large amount of plaintext until the output matches the ciphertexts he has.
- Ciphertext-only based attack. This means the attacker only has a set of ciphertext.

In the case where an opponent knows in advance the set of programs that could be used, many approaches could be used to determine which program has been transmitted. For example, the length of the encoded program provides information that may be sufficient for this type of attack. Hence our approach would not be suitable for distributing only a small set of known programs.

Another attack that could be considered is using statistical information about where specific bytecodes tend to occur in programs. Should the distribution not be uniform, this would provide information that could be used to infer parts of the mapping. We plan on extending our approach to consider these issues as well.

A useful extension of this work would involve using multiple bytecode translation tables. Translation could switch between tables following an agreed upon pattern. The effective key for this encoding would be the set of bytecode mappings and the pattern used to choose the table in effect at any given moment. This would effectively transform the approach into a type of block cipher. This should be feasible when enough storage is available and use approximately the same amount of resources as the approach described here.

Note that our approach is presented as a form of data obfuscation. We feel that this is an appropriate label since it provides some protection against the information being accessed by unauthorized parties, but is less secure than standard encryption techniques. It is appropriate for use in resource-constrained environments, where encryption would be impractical. When the information is valuable enough and resources are available, encryption methods such as RSA or AES would be a more appropriate choice.

On the other hand this approach is probably more secure than techniques in widespread use, like the use of hash functions. The approach proposed is more secure than many obfuscation approaches, because it is not sufficient to know the algorithm being used. The unauthorized user must still find a way of determining the bytecode mapping. In this way, the bytecode mapping itself could be considered a cryptographic key. The approach proposed can also easily be integrated with other Java obfuscation techniques, such as control flow obfuscation.

The simplest substitution method is called mono-alphabetic substitution cipher. This method substitutes a letter into a fix symbol or letter. This kind of method is vulnerable to every cryptanalysis attack described in the last section. To reduce the vulnerability, we can use more than one permutation table called poly-alphabetic substitution cipher. The permutation table that is used to map a letter depends on the position of that letter. This helps to balance the frequencies of alphabets and makes frequency attacks more difficult than using just mono-alphabetic substitution ciphers. However, when using poly-alphabetic substitution ciphers, an attacker with the knowledge of the number of permutation tables and the usage order of these tables will be able to break these codes. In these cases, frequencies of pairs or triplets of alphabets are used to determine the number of permutation tables and usage order.

Our approach to bytecode substitutions is based on applying the ideas of alphabetic substitution for secure code and transmission. In our bytecode substitution approach, multiple substitutions are applied one after another in order to make frequency attacks difficult. Further, the use of instruction folding helps in balancing the frequency of sequences of bytecodes (note that frequency attacks on sequences are a major concern in the case of poly-alphabetic substitution).

With the continued proliferation of mobile devices, it is becoming important to address many of the issues associated with programming these devices. In particular, there has been a rapid growth in embedded Java devices that have become attractive due a variety of reasons such as platform independence, on-demand loading and compilation and remote update/execution. Main concerns in remotely programming mobile devices are in ensuring that only authorized users can execute the code and that the device cannot be harmed by malicious codes. In this work, we focus on the first aspect and use various substitution based mapping schemes to provide a computationally less intensive alternative to using standard encryption. Our experiments with 52 Java class files reveal that our approach is very effective in protecting the code being transmitted. All the substituted codes failed in the verification phase of the unauthorized devices that did not have access to the substitution table. Further, our analysis reveals that careful creation of the substitution table will also make it very expensive for frequency based attacks to be successful.

6. MODES OF NETWORK BEHAVIOR

In the previous sections we have discussed:

- Models of networked systems as a set of interacting semi-autonomous systems.
- How to design and implement loosely coupled systems with desirable global properties.
- How to secure individual components in the global system.

This section now considers large-scale epidemic attacks. First, we present techniques for detecting attacks that build on insights from our modeling work. We then discuss an initial game theoretic analysis of where the vulnerabilities exist in large-scale distributed systems. We end this section by showing how the current network topologies are particularly friendly to worm and virus attacks. The work in this area just scratched the surface of what needs to be done to make networks resilient to attack.

6.1 DETECTION OF DISTRIBUTED DENIAL OF SERVICE (DDOS) ATTACKS

The majority of network-based DoS attacks involve an excessive number of packets directed at the victim machine. So-called "flood" attacks exist for TCP, UDP, and ICMP protocols to name a few. The most common DoS attack is the TCP SYN flood, and we use this attack to test the DoS detection method. The SYN flood exploits the three-way handshake TCP uses to establish a connection. The attacker sends connection requests with forged source addresses to a server. The server stores these false connection requests while it tries to complete the connection. Each request is stored until either it is completed or it times out. Since these connections can never be completed, the server's queue fills waiting for responses. Legitimate users are then unable to initiate connections.

Other flood attacks tend to have properties similar to the SYN attack. While the rest do not have the advantage of the TCP connection handshake, the underlying concept is the same: keep the server's queue (and routers' queues as well) full enough that non-malicious users cannot connect.

In the past, many of these attacks could be performed through small bursts of packets because queue implementations were not robust and timeouts were needlessly long. Today, networking implementations are more robust so that these flood attacks must generate an inordinate amount of packets to have the same effect. Because of this, most attackers have moved to a Distributed Denial of Service (DDoS) attack where many slave "zombie" computers are used to attack a victim. Zombie processes can be planted far in advance of the planned attack. This results in a broader, less localizable attack that is inherently more difficult to detect and defend.

A packet flooding based DoS attack typically results in a sudden, serious change in one or many of the victim's system or network resources. These changes may not be easily discernable, but would include things like the length of the server's backlog and connection queue, the total number of incoming packets as well as the number of specific types of packets, and the disparity between the numbers of different components of the TCP connection handshake.

Internet traffic is highly dynamic. Both the mean and variance of incoming packets are time-dependent and are expected to have a large variability. After an attack begins, the traffic mean and variance will suddenly increase as the traffic due to the attack is superimposed on the original traffic. Measuring changes in such a highly dynamic process is a complex undertaking.

The DoS detection approach we are testing monitors the number and type of incoming IP packets over time. To verify the robusness of this approach, we apply it to the following:

- Network simulators (which frequently do not have the ability to break apart the TCP handshake)
- Recorded real-world data
- On-line testing in our laboratory

Other system and network resources have properties similar to the number of incoming packets. Applying concepts similar to the our DoS detection approach to monitor other network protocols, or quantify, measure, and analyze internal systems resource availability should be straightforward.

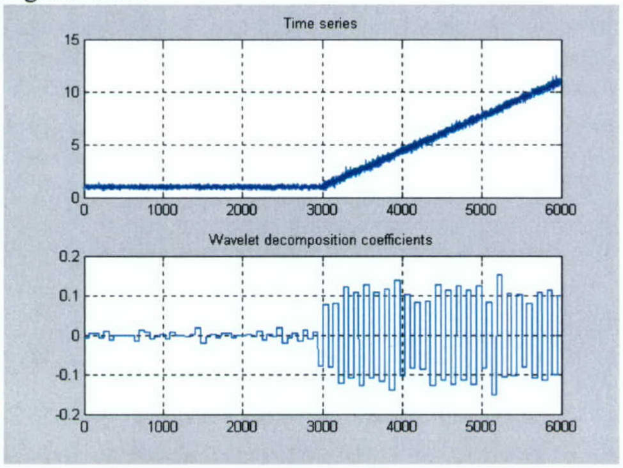


Figure 58 (Top) – The CUSUM statistics of the model of an arriving packet stream with a DoS attack starting at 3000 seconds; (Bottom) – The Wavelet Coefficients at the 6th level of decomposition.

To detect when the DoS occurs, let N(k) denote the number of packets received during the kth time interval. To reduce the noise of N, we perform a moving average on N giving $\widetilde{N}(k)$. The following modified cumulative sum (CUSUM) function is applied to this averaged data:

$$\widetilde{S}(k) = (\widetilde{S}(k-1) + \widetilde{N}(k) - m - c)^{+},\ \widetilde{S}(0) = 0,\ c > 0,$$

The modified CUSUM statistic features a nonlinear operation + that returns the greater of (x,0). This combined with the subtraction of the expected value m and a small constant c, reduces the variance of the statistic before the change. It has little effect on the variance after the change. The top image in figure 58 shows the modified CUSUM statistic for an idealized DoS attack. The idealized attack is a step function corrupted with Gaussian noise.

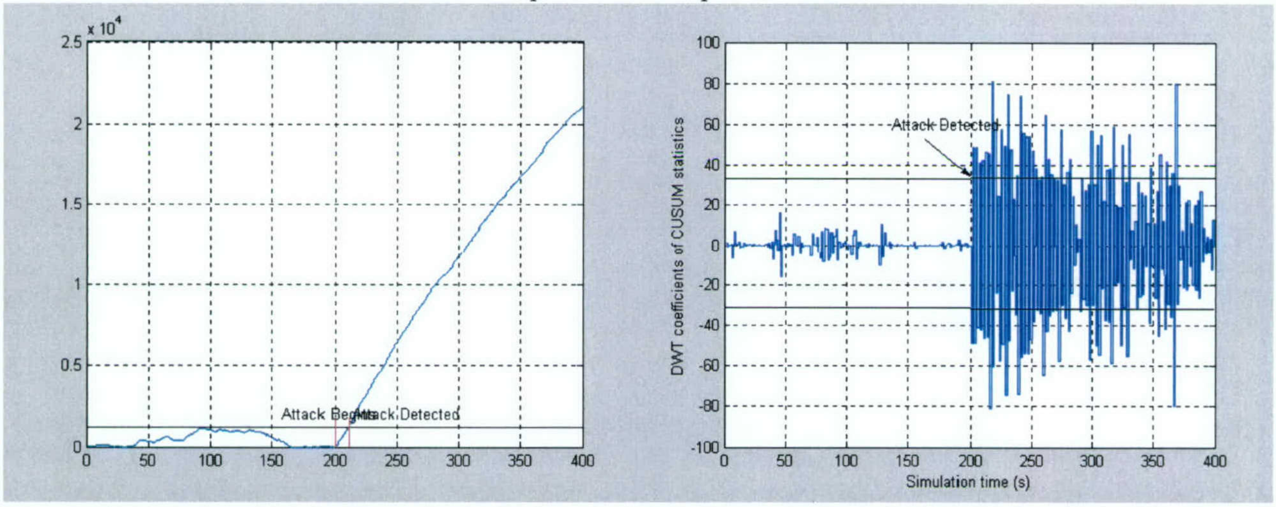


Figure 59 – Attempting to detect the attack using a CUSUM-only method on NS-2 data (left) Detecting the attack using the wavelet decomposition of the CUSUM method (right). Notice that the delay is much small than figure 58.

To further highlight the change point of this CUSUM statistic, we perform wavelet analysis using the discrete wavelet coefficients of this statistic at the 6th level of decomposition. Wavelet decomposition at this level is essentially a high-pass filter, since change points are high frequency items. For this application, we use the Haar wavelet. The bottom image in figure 58 is the Haar wavelet coefficients at the sixth level of decomposition of the top image.

A DoS attack on a computer results in an increased mean and variance of the arriving number of packets. Given this increase, the CUSUM algorithm should also show a substantial and sudden increase. At the 6th level of wavelet decomposition, this change is made apparent via coefficients on the order of 4 times larger than under normal traffic conditions. Using this method to detect DoS attacks results in detection at an average of 1.13s after attack (versus 7.6s for the CUSUM-only method). Figure 59 illustrates this, using data from an NS-2 simulation.

Wavelet analysis of the CUSUM has better detection efficiency of DoS attacks than the CUSUM approach alone. The increase in detection ratio is 56%. Even higher detection ratios are possible with higher levels of wavelet processing. The first stage of CUSUM processing lowers the amount of noise of the arrival process, while the subsequent wavelet analysis finds the change-point of the time-series. The change-point is a DoS attack.

The Haar wavelet was selected for its simplicity and previous application for change-point detection. Other mother wavelets functions should be evaluated. Better detection efficiency, delay, or other performance gains may be achieved. Application of wavelet analysis directly to the network time-series should also be researched for completeness.

Different sources of test data were used. Various synthetic data sources were investigated, but reliance was placed on captured live data. Realistic data provides better confidence in test results. Lacking availability of a captured DoS attack, a simple attack model was superimposed on live traffic of high variance and rate. Live traffic dynamics will present more difficulties to the anomaly detection systems, irrespective of the DoS model. Therefore we suggest our DoS modeling is reasonable when superimposed on Internet 'noise', but not absolute. More accurate approaches to DoS attack modeling should be explored.

It appears that our detection approach could be placed anywhere in the network. Placement at natural chokepoints, like firewalls, is likely to be a good strategy.

Detection results from the NS2 and live data sets were noticeably different. NS2 was ideal over a large range of parameter settings. This is due to the synthetic nature of the NS2 test data, which does not contain enough background traffic variability. Live data does not obtain ideal detection efficiency over any set of parameters settings. The background traffic of the live data set challenges the detection system, thus reducing its true detection rate and providing a non-zero false positives rate. Tradeoffs in detection efficiency is possible through parameter 'tuning' as indicated in the ROC graphs of section VIII. Each has an effect of removing various amounts of arrival process noise or burstiness from the network time-series. This increases the signal-to-noise ratio, allowing wavelet analysis to detect the abrupt change due to the DoS attack. Detection delay is dependent on amount of wavelet processing. Low wavelet decomposition levels (*WDL)* provide better detection delay than a purely CUSUM approach, but at higher levels of wavelet analysis, the delay may increase. A tradeoff exists between increasing the *WDL* for detection efficiency gains and possible detection delays losses. Further iterative testing on independently created datasets should be performed to ensure accuracy and consistency of tuning parameters.

DoS flooding attacks cause significant changes in the amount of network traffic. Similarly, regular network activity can have large variations. Traffic fluctuations can occur from topology changes, user activity, or network management. Examples include flash crowds, data backups, network maintenance, and administrative changes. A regular network event was declared as a false positive in several live traffic time-series. This recurring false positive event is shown in Figure 60 at interval $k = 1.3x10^4$ and $7.3x10^4$. This regularly timed event, possibly a data backup, mimic a DoS attack through a large packet count increase. Such events should be analyzed more closely for better understanding and modeling. The wavelet-based DoS detection method in its current form cannot distinguish these events as normal activity. New detection rules or algorithms are needed to avoid these false positives.

For a live dataset, 78% true detections, 37% false positives, for a detection ratio of 2.1 was determined. Over the 239 time-series, each of which on average is 8 hours in duration, the false positive rate equates to 1.11 per day. Although high, approximately five false positives per week are due to our network's 'regular' events described above. For a deployable solution, future improvements are needed, but they should be flexible, as detection of these normal events may be of some interest.

Other malicious network events need to be modeled (or captured) for detection evaluation. Viruses, worms, and slow-start DoS attacks are examples. Once detected, attack response countermeasures may be effective or could cause further problems. For example, packet blocking from a subnet is a common response to a DoS attack. Blocking removes both attack and legitimate traffic. If the detection's threshold is set too low, the packet filter can be engaged prematurely and obstruct legitimate traffic. Adequate response to attack detection is an open issue.

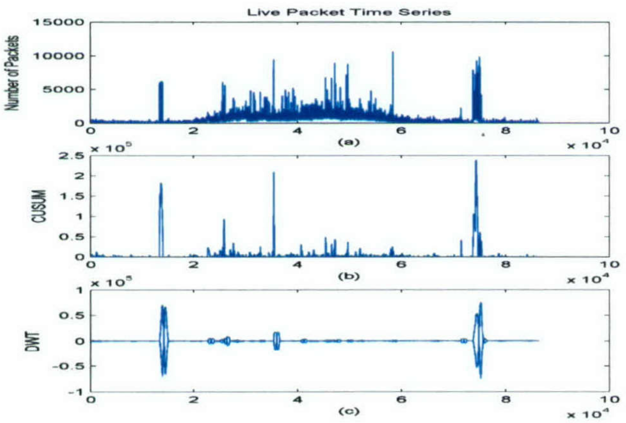


Figure 60 – Live Traffic False positive. (Top) Live Packet Time Series; (Middle) CUSUM Statistic; (Bottom) Wavelet Decomposition Coefficient ($d_{10,0}$).

Reasonable computational complexity of the wavelet-based algorithm allows for on-line implementation. The CUSUM algorithm of (8) requires little memory and involves simple arithmetic operations. Depending on the operating environment, alterative implementations for the windowing (7) and estimation functions (6), less floating point multiplication, can be sought. Wavelet analysis by the discrete wavelet transform has a memory requirement of 2^j. For wavelet decomposition levels of j = 6 or 10, this is a low $2^6 = 64$ or $2^{10} = 1024$ units of memory. Computation of the wavelet coefficients through Mallat's pyramid algorithm requires only elementary operations of addition, subtraction, and shifting. All algorithmic computation can be easily implemented in either software or hardware.

DoS attack detection is possible through analysis of network time-series. The use of both statistical and wavelet analyses provides better performance than a purely statistical approach. Higher true detection and lower false positives rates are seen. Detection delay can be reduced or adjusted at the expense of detection efficiency. Use of realistic network traffic ensures confident in the test results.

6.2 DDoS PREVENTION

We model a network as a graph $G = (V, E)$, where V is a set of vertices (or nodes) and $E \subseteq V \times V$ is a set of edges (or links). The structure of the edge set E describes the connections that exist between nodes in the network. A distributed program P, consisting of programs $\{p_1, \dots, p_k\}$ running in the network described by G, can be represented by assigning each program of P to a vertex of G via an assignment function $c : P \rightarrow V$. Each distributed application has a certain set of connectivity requirements that the network must satisfy in order for the distributed program P to execute successfully. For example, suppose that P is a distributed client/server system. Then each client must be able to connect to the server, however it may not be necessary for the clients to be able to connect to each other. We formalize this notion by saying that a program p_i is connected to a program p_j if there is a path connecting the vertex containing p_i with the vertex containing p_j. This sentiment may be formalized by a sentence π_{ij}, written in the first order graph predicate language. We shall denote by Π the set of all communications requirements for the distributed application P. When a network G and an assignment c satisfy all the requirements of a distributed application P, we shall write $(G, c) \models \Pi$, where it is understood that Π is the set of requirements of P.

Let $G = (V, E)$ be a graph describing a network, P be a distributed application, $c : P \rightarrow V$ be an assignment of programs to vertices and Π be a set of requirements such that $(G, c) \models \Pi$. A successful distributed denial of service attack (DDoS) transforms the graph G into a new graph G' such that $(G', c) \not\models \Pi$. DDoS attacks work by introducing zombie programs into the nodes of G. These programs produce spurious packets, which either cause traffic congestion within the network itself or render a node in the network unavailable to receive legitimate traffic by bombarding it with illegitimate connection requests. To model the susceptibility of vertices and edges to zombie traffic, we defined a function $\tau_V : V \rightarrow \mathbb{N}$ ($\tau_E : E \rightarrow \mathbb{N}$) giving the number of simultaneous zombie attacks a given vertex (edge) could sustain before being rendered operationally ineffective by the attacker. In this model, we assumed that all zombies produce malevolent traffic at an equal rate and strength 1.

Brooks and Griffin have studied the following problems related to this idealized view of DDoS attacks:

1. What is the minimum number and configuration of zombies necessary for red to disrupt a requirement $\pi \in \Pi$?
2. What is the maximum number of requirements that can be disrupted by the attacker?
3. What is the minimum number and configuration of zombies necessary for red to disrupt the greatest number of requirements in Π?

[1]This is not necessarily true, since a zombie's ability to produce traffic is related to the properties of the computer it is using. However, for our purposes, it is safe to assume that all nodes are created equally in terms of their ability to produce malevolent traffic when housing a zombie.

The main result of our investigation was the following theorem, which completely determines the security of a distributed application running in an idealized computer network.

Let G be a network and and P be a distributed application and let $c : P \to V(G)$ be the position of the programs in the network. Furthermore, let π be a requirement in Π saying that v_1^π must be connected to v_2^π. Then there is an algorithm with low order polynomial running time that determines whether π can be perpetually disabled; i.e., whether there is an attack such that for all counter-strategies the resulting graph $G' \not\models \pi$.

To prove this result, we defined the minimum security edge cut and minimum security vertex cut of the graph G. Removing the edges and vertices in these sets will disconnect a graph. However, unlike the minimum edge (vertex) cut, which has the least number of edges (vertices), the minimum security edge (vertex) cut requires the smallest number of zombies of any other edge (vertex) cut to disable the edges (vertices) in the cut. Using this formalism, we were able to reduce the problem of determining the minimum security edge (vertex) cut to a max flow min cut problem. Having shown this, we derived an algorithm to disable a single edge or vertex in the network using the smallest number of zombies . We proved this algorithm was minimal in its zombie use by applying a second min-cut arguement. Taken together, these two results yield theorem . The running time of the algorithm is at worst square in the edges and vertices of the graph, since this is an upper bound on the running time of the pre-flow push algorithm used to determine min-cuts in networks.

We can immediately use this result to help us determine optimal node placement in ad hoc mobile networks and optimal software placement in mobile code networks. Consider the later problem, when we are given mobile programs $P = \{p_1, \ldots, p_n\}$ and must determine an optimal placement for these programs under changing network conditions. In particular, suppose we are given a set of new criteria Φ, consisting of sentences $\phi_1, \ldots, \phi_k$ each stating that some program p_i must be placed in some subgraph H of G. The next result follows immediately from above.

Let G be a network and P be a distributed mobile application. Let Π be a set of connectivity requirements for the programs of P and let Φ be a set of positioning requirements. Then there is a function $c : P \to V(G)$ that will minimize the maximum number of requirements of Π that can be perpetually disabled by DDoS attack. Furthermore, this function can be computed in $o(|P|^G)$ steps.

This optimal placement function can be computed by using the results from the proof of theorem , were we compute minimum security edge and vertex cuts for the entire graph. Since this placement is an absolute minimum, it is clear that it will not vary in the presence of optimal attacks; i.e., there will be no reason to recompute c, assuming that an attacker is playing optimally.

It is not clear whether it is possible to recover c using a polynomial algorithm. However, it may be possible to use a genetic algorithm with the fitness function being the security of the network and the population being placement functions of the nodes. More research must be done on this question to determine the best way to find the placement function c in the presence of a dynamically changing network.

Our results can be extended to mobile ad hoc network security questions as well. Brooks et al. have derived a formulation for the expected structure of a random graph, given the probability distribution governing its structure. Their algorithm runs in polynomial time. Hence, we may apply our theorems in the case of the expected value to find the expected vulnerability of a given graph to attack. The following theorem follows immediately:

Let $\mathcal{G}$ be a random graph family with a fixed number of nodes and let P be a distributed mobile application. Let Π be a set of connectivity requirements for the programs of P and let Φ be a set of positioning requirements. Then there is an expected function $c : P \to V(G)$ that will minimize the maximum number of requirements of Π that can be perpetually disabled by DDoS attack. Furthermore c can be computed in at worst $o(|P|^G)$.

We studied DdoS attacks using the *ns* network simulator. An ns-2 simulation script was written to generate pseudo-random traffic between connected nodes. Link speeds and delays were determined by finding the degrees of the connected nodes. The bandwidth of nodes of each degree are shown in table I. Both the speeds and propagation delays of the links were scaled down to unrealistic speeds of arbitrary determination due to the infeasibility of simulating the thousands of nodes needed to generate realistic network traffic along links at actual Internet speeds. The link speeds and propagation delays ranged from 10 Mbps to 233 Mbps and 10 ms to 40 ms, respectively. The reasoning for choosing increasing speeds and delay times with increasing degree was based on the assumption that the number of connections from a node is proportional to the importance and rarity of the device.

The topology that was chosen for the simulations emulated a medium-scale network. Forty-one nodes were available to host zombies and fifteen nodes to be victims. In testing, only nodes with a single link could host zombies, while victims were restricted to core nodes not directly connected to a zombie node. The maximum hop distance, assuming that no cycles occurred, was nine hops. All connections were established as point-to-point to simplify the analysis of the packet flows. Any leaf node was assumed to be a point at the edge of an internal LAN and thus was modeled appropriately with our link speed and propagation delay definitions. The multipath option for ns was enabled for the topologies to allow packets to take multiple routes through the core nodes to reach the destination.

All background traffic used TCP since it is the de facto standard of the transport layer on the Internet. For burstier, non-uniform traffic, a Pareto random variable was used to approximate background Internet traffic because of their heavy tail and infinite variance. Conversely, for a more consistent pattern of traffic, the constant bit rate (CBR) generator was used. For both patterns, the TCP Reno agent was selected for the sender. The selective ACK sink was used to decrease the number of ACK packets the receiver must generate to alert the sender of a successful transmission.

To choose the sender and receiver pairs for the background traffic, a time-seeded uniform random variable was created to generate the integers representing the nodes. For our topology, the senders were restricted to the nodes between 0 and 40, while the receiver could be any node in the network. One hundred random pairs were generated with the guarantee that at least four connections would be created between four random senders and the victim. Except for tests involving background traffic structure, the one hundred sender and receiver pairs were kept the same for all simulations.

The DDoS attack was a link flooding attack generated by zombies placed at specific nodes in the network. A UDP agent was attached to a zombie with a CBR traffic generator. UDP was chosen for the zombie packets because of its connectionless characteristics. The CBR generator was configured to send a 404 byte packet every 0.5 ms. Note that the zombie packets are significantly smaller and sent more frequently than legitimate packets, as expected with a DoS attack.

To generate the order in which to add zombies, another time-seeded uniform random variable was created. This list was generated before the simulations and held constant through all simulations except for the zombie placement tests.

To track the success or failure of the DDoS attack, multiple methods were used dependent on the scope of the analysis. When considering the entire network, queue traces were placed on links connected to "sending" nodes. This allowed us to calculate the total number of legitimate packets sent for all leaf nodes. Queue traces were also placed on all of the receiver's links, allowing for a count of the number of received legitimate packets. The traces were configured to record all link activities. After completing each simulation, the traces were parsed to calculate the number of legitimate packets sent and received as the DDoS attack increased in strength.

To analyze the success of the DDoS attack on a specific node, traces were placed on links connected to nodes with a session to the victim. Additionally, all links connected to the victim were monitored. The packet ID fields in the ns-2 traces were parsed and compared to determine if the packet arrived at the victim successfully. Note that this does not take into account any packets that are routed through the victim, which would be contained within the first method of analysis.

A subset of all possible tests was simulated with varying parameters. To test the effect of traffic style on a network, the placement of zombies and the sender/receiver pairs were held constant. Simulations were then performed on three different zombies with both types of traffic, CBR and Pareto. The number of zombies was then varied from 1 to 41. The same experiments could then be varied by changing the zombie locations and background traffic pairs. It should be noted that the numerical values of the packets and network were not our focus in this research, only the general patterns associated with performing a denial of service attack upon a test topology that would resemble an actual network.

In the first simulations, we analyzed the combination of the traffic flows in the network. The data indicates that when not under a DDoS attack, the flows behave consistently and do not suffer much, if any, loss. This loss can be reasoned with TCP window size and congestion control effects. We assume that the ns-2 implementation of TCP follows the behavior of the "real world." As more individual sessions generate traffic, TCP window size will increase until the network is congested. This will cause a small packet loss if the simulated routers' queues are full. At this time, our data indicates that the congestion control algorithms of TCP are used to decrease the window sizes. On average, our simulation setup experienced a 0.04% loss rate due to only background traffic.

When a DDoS attack was overlaid on the background traffic, similar packet loss occurred but at a much larger scale. A total of twenty-two tests were performed per victim with different patterns of background traffic and zombie addition. For each case, the number of zombies was increased by one and the loss recorded from the addition of the extra zombie. Zombie positions were chosen from a uniform distribution ranging from [0,40]. Simply placing zombies into the network did not always increase the loss experienced by the flows. In some cases, the resulting loss would either be zero or negative, *i.e.* the network loss rate decreased. This indicates that the addition of that particular zombie created a less optimal network state, with respect to the zombies. The traffic generated by the added zombie interfered slightly with the preexisting background and zombie traffic.

In Figure 62, the addition of zombies 30 through 32 caused the loss rate to decrease. When these particular zombies were moved to the end of the addition list (they were the last to activate), the loss rate did not experience the drop it had when the moved zombies were activated earlier in the sequence. If the zombies were moved to the beginning of the addition list (they were the first to activate), the network did not experience the maximum loss percentage that the original sequence underwent.

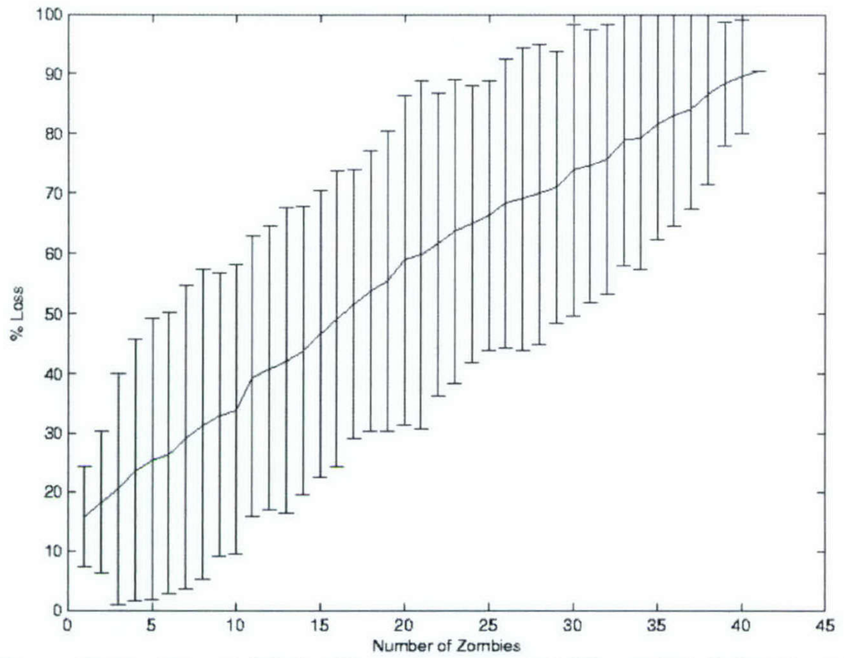


Figure 61: Legitimate Packet Loss % vs. Zombie Count (Node 44 Pareto)

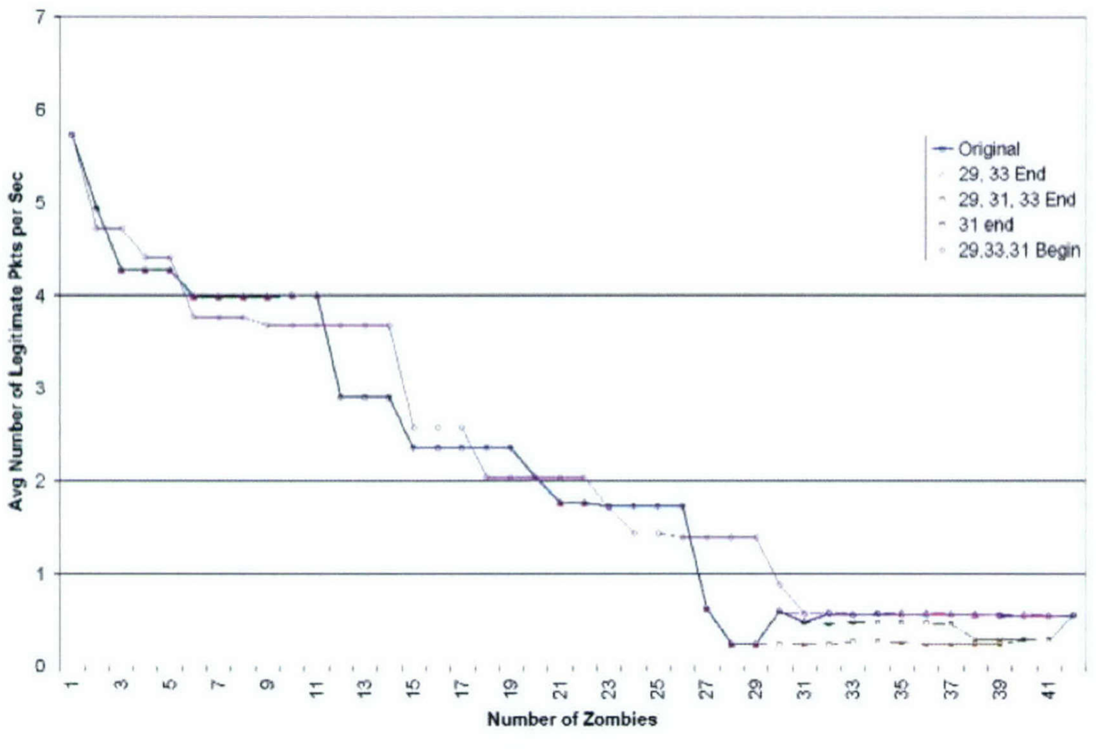


Figure 62: Legitimate Packet Loss % vs. Zombie Count (Node 44 Movement)

Suppose that when operating at maximum network efficiency, the victim can receive $r(t)$ packets when no zombies are present. We will attempt to construct an expression for the *measure* of the DDoS attack from $r(t)$. Let $\gamma(t)$ be the number of legitimate packets generated by time t and suppose that n is the number of zombies placed in the network. If each zombie generated $\zeta(t)$ packets by time t, then we can estimate the number of legitimate $y(t)$ packets received at the victim by time t as:

$$y(t) = \frac{r(t)\gamma(t)}{\gamma(t) + n\zeta(t)}.$$

Computing the ratio of legitimate packets received to legitimate packets generated, we have:

$$s_D(n) = \frac{r(t)}{\gamma(t) + n\zeta(t)},$$

where $s_D(n)$ is the success-rate of the DDoS attack, when we assume that $\gamma(t)$, $r(t)$ and $\zeta(t)$ are linear in t. Figure 63 shows the mean s_D value over 23 runs using node 44 as the victim, our estimator of $s_D = 65.182/(65.182 + 1.004n)$ and also a linear regression on the data set $s_D = 0.010n + 0.020$ used to produce the mean. While the linear regression was more accurate, with a SSR (sum of squares of residuals) value of 5.41, our predicted regression has a SSR value of 5.61 and more is more reasonable as a model. Furthermore, the variance of the data was high as is shown by the standard error computed around the mean.

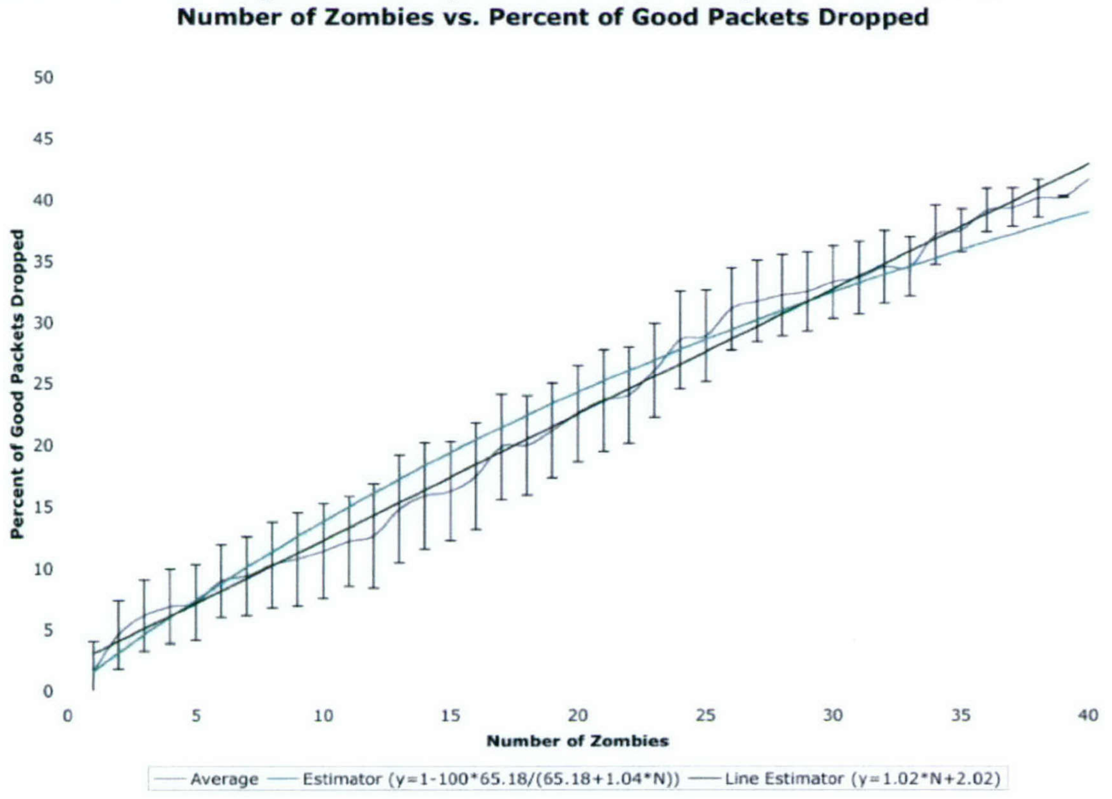


Figure 63: A linear regression, non-linear regression and the average of a data set of attacks on node 44 with error bars

6.3 WORM AND VIRUS PROPAGATION

We have studied the propagation of worms and viruses in Internet like networks, building on the work of Aiello *et al.* We begin assume that the Internet is a scale-free graph G with some scaling parameter $\gamma>2$. In this case, Aiello *et al.* showed that almost surely the internet has a connected component with size $O(|G|)$ just in case $\gamma<\gamma_0\sim 3.4875\ldots$ The value of γ_0 arises from the solution to an equation involving ζ(□)□□the Riemann-Zeta function. If we assume that each worm or virus only requires a connected graph to propagate, then we arrive at the following conclusion:

If G is a scale-free graph with scaling parameter $2<\gamma<\gamma_0\sim3.4875$, then every virus will become epidemic within the network; i.e., eventually every node in the network will become infected.

This of course does not take into consideration the possibility that only a proportion p of the population is susceptible to the pathogen. Unfortunately, the scale-free graph model we used leads to the following conclusion:

If G is a scale-free graph with scaling parameter $2<\gamma<\gamma_0\sim3.4875$, then any pathogen that infects a proportion $0<p\leq1$ of the population will become epidemic within the susceptible population; i.e., every susceptible node in the network will become infected.

Furthermore, the propagation time required for complete infection can be shown to be very low, in fact Aiello *et al.* showed that the diameter of G is $O(\log|G|)$, thus allowing us to conclude that a very small number of hops must be taken by each virus or worm before complete infection is guaranteed. Specifically, suppose that a virus spreads with speed v nodes per second. If this speed is constant (and does not diminish as more nodes are infected), then the number of infected nodes will grow exponentially quickly. This result was confirmed anecdotally by the nature of the Slammer worm spread in MS SQL Servers. However, it is worth noting since worms seek out uninfected hosts, their propagation often resembles a logistic curve, instead of a pure exponential curve.

We also found that for scale-free networks with $\gamma>\gamma_0\sim3.4875$, there are never epidemics. In particular, Aiello *et al.* proved that in the case when $\gamma>\gamma_0$, then the largest component in G was of size $O(\log|G|)$. Thence, viruses cannot spread because the network substrate is not sufficiently connected to support viral propagation beyond a small region near the initial infection point.

Using these facts, we derived a number of non-traditional counter-worm measures that we believe could be quite successful. Since worm epidemics are assured in existing network structures, where γ much less than 3, there is little hope of ever patching a network quickly enough in response to a virus attack. A patch solution requires each susceptible node to be updated. The limitations of bandwidth and mirror sites for obtaining this patch lead us to conclude that patch solutions run in $O(|G|)$, much slower than the speed of propagation of a virus. Conversely, if we introduced a white-worm (an engineered worm designed to seek out infected and susceptible systems for the purpose of repair), then repair and patching actions could be accomplished in $O(\log|G|+C)$, where C is a constant of proportionality that depends on the amount of time that has past since the pathogen has been released into the wild till a white-worm was released. Our theoretical result has been validated experimentally by Chen and Carley [CC04]. They showed that the most efficient system patching technique was counter-worm patch propagation.

We also observed that changing the value of γ will also stop worm propagation in networks. Such changes do not have to be made at the hardware level. New protocols can be introduced into the Internet that will effectively raise the value of γ. For example, suppose that we require a trusted third party certification to allow network traffic to leave local area networks. This third party would be required to validate out going traffic. Clearly, such a solution might be cost prohibitive to construct, but it would achieve the desired effect. Worm traffic would have to be certified by a third party. This means that worms would face a higher cost to spread than they presently do. Since this affects all network traffic, the result is the effective fracturing of the Internet into local sub-networks with a higher cost (in terms of time, trust and possibly cash), for accessing global services.

7. DISCUSSION

The MUSE project performed basic research on the topic of understanding mobile code. It made significant advances in this area. Mobile code differs from other software systems in that it uses networks to autonomously move code from one host to another.

Many common CIP threats, such as Trojan horses and viruses, pre-date widespread use of the Internet and are not specific to mobile code. Issues such as insuring program correctness, enforcing security policies, avoiding buffer overflows, and detecting malicious code also exist for non-networked software. Our emphasis is on researching how code migration affects infrastructure protection.

Viruses, worms, and Denial of Service (DoS) attacks are difficult to counteract in large part because they are highly distributed. Fortifying the defenses of individual processors, or even sub-nets, cannot sufficiently neutralize these threats. Our game theory analysis of DoS attacks contains examples of the limitations of firewalls for protecting distributed systems. Fortifying individual processors is in some ways similar to building a stronger Maginot line after World War II.

MUSE studied both the threat posed by malicious mobile code, and the promise of mobile code to adapt when attacked and neutralize threats. Distributed adaptation can put attacked systems on an equal footing with their attackers.

The project Statement of Work (SoW) consisted of four tasks:

- Develop a theoretical model
- Study the interface between mobile code and the host computer.
- Study system adaptation
- Create an adaptive network infrastructure.

Significant results include:

- A theoretical model for mobile code was developed by integrating mobile agent and cellular automata concepts. Using a simulation tool (CANTOR), that we developed for this model, we found important behavioral differences among mobile code paradigms.
- CANTOR simulations were found to trend like other network simulators. The CANTOR models are simpler and contain fewer factors. They also execute more quickly than traditional approaches.
- A taxonomy of mobile code paradigms was created combining our theoretical model with an existing taxonomy of network attack vulnerabilities.
- Our existing mobile code daemons were integrated with peer-to-peer (P2P) indexing to create an initial adaptive infrastructure. Cryptographic key management has been integrated into this approach.
- We proved that cryptographic primitives can be used with a tamper-proof co-processor to verify bilateral trust between a host and a mobile code package. An application to multi-level security was given.
- Quality of Service (QoS) analysis of peer-to-peer (P2P) networks has been performed to derive the proper numbers of indexes and packet time-out values to balance system performance and robustness.
- Our mobile code and P2P infrastructure was used to create a proof of concept distributed system that adapts around DoS attacks.
- A random graph model has been developed for P2P systems. This model predicts the expected robustness and performance of P2P networks created by nodes connecting with given statistical patterns.
- We defined DoS a game theoretic problem. We then analyzed player strategies to measure network vulnerability. Among other things, results of this work show both the utility and limits of firewalls for DoS protection.
- We created a method for early detection of DoS attacks in conjunction with Dr. Rai of LSU. This method has been validated using simulations, DoS attacks in the laboratory, and live network data collected at PSU.

- Implementation of a secure instruction set that can foil power analysis attacks. We have shown how this approach can protect DES keys stored in smart cards from being deciphered by using differential power analysis attacks.
- Found the relationship between network topology and virus propagation. This leads to clear suggestions as to how to best avoid and respond to worm and virus attacks.

8. CUMULATIVE LIST OF PUBLICATIONS SUPPORTED BY THIS GRANT

The following publications attributed to RSN have been published, are under review, or are in press:

Research Monographs:

- ***Disruptive Security Technologies with Mobile Code and Peer-to-peer Networks*** by R. R. Brooks, to be published by CRC Press in 2004. Contract signed. Book in preparation.

Peer-reviewed journal publications:

1. J. M. Zachary and R. R. Brooks, "Bidirectional Mobile Code Trust Management Using Tamper Resistant Hardware," *Mobile Networks and Applications*, 8, pp. 137-143, 2003.
2. R. R. Brooks, and N. Orr, "A Model for Mobile Code using Interacting Automata," *IEEE Transactions on Mobile Computing*, Vol. 1, No. 4, pp. 1-14, October-December, 2002.
3. H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. R. Brooks, S. Kim, and W. Zhang, "Masking the Energy Behavior of DES Encryption," Proceedings of *IEE*, Accepted for publication, June 2003.
4. R. R. Brooks, N. Gautam and C. Griffin, "Games on Graphs for Modeling Distributed Denial of Service Attacks," *ACM Transactions on System and Information Security*, submitted for review, July 2003.
5. T. Keiser, and R. R. Brooks, "Implementation of Mobile Code Daemons for a Wired and Wireless Network of Embedded Systems," *Internet Computing*, in press, May 2004.
6. M. Young, J. Schwier, R. R. Brooks, and S. Rai, "Testing Denial of Service (DoS) Detetection Methods," *Internet Computing*, revised, September 2003.
7. R. R. Brooks, S. A. Racunas, and S. Rai, "Mobile Network Analysis using Probabilistic Connectivity Matrices," *IEEE Transactions on Mobile Computing*, under revision, June 2004.
8. R. R. Brooks, C. Griffin, and A. Payne, "A Cellular Automata Model can Quickly Approximate UDP and TCP Network Traffic," *Complexity*, in press, May 2004.
9. R. R. Brooks, "Mobile code paradigms and security issues," *Internet Computing*, In Press, April 2004.
10. R. R. Brooks, C. Griffin, and J. Schwier, "What makes a distributed denial of service attack work well?" *IEEE Transactions on Dependable and Secure Computing*, submitted for review, May 2004.
11. G. Carl, R. R. Brooks, and S. Rai, "Wavelet-based Denial of Service Detection," IEEE Transactions on Computers, submitted for review, May 2004.
12. R. R. Brooks and C. Griffin, "A note on the spread of worms in Internet-like networks," *ACM Transactions on and Information Security*, submitted for review, May 2004.

Conference publications:

13. H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the energy behavior of DES encryption, *Proc. the 6th Design Automation and Test in Europe Conference (DATE'03)*, Munich, Germany, March 2003. (Nominated for Best Paper)
14. A. Kapur, N. Gautam, R. R. Brooks, and S. Rai, "Design, Performance and Dependability of a Peer-to-Peer Network Supporting QoS for Mobile Code Applications," Proceedings of the *Tenth International Conference on telecommunications systems*, pp. 395-419, Sept. 2002.
15. R. R. Brooks, C. Griffin, J. Zachary, and N. Orr, "An Interacting Automata Model for Network Protection," Invited Paper, *Fusion 2002*, July 2002.
16. R. R. Brooks and C. Griffin, "Fugitive Search Strategy and Network Survivability," *2003 Industrial Engineering Research Conference*, invited Paper, January 2003.

17. E. Swankoski, R.R. Brooks, V. Narayanan, M. Kandemir, and M. J. Irwin, "A Parallel Architecture for Secure FPGA Symmetric Encryption ," *Reconfigurable Architecture Workshop*, Santa Fe, New Mexico, April 2004.
18. H. Saputra, N. Vijaykrishnan, M. Kandemir, R. Brooks, and M. J. Irwin. Exploiting value locality for secure energy aware communication. In Proc. the *2003 IEEE Workshop on Signal Processing Systems (SIPS'03)*, August 2003.
19. G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R.Chandramouli. Energy-aware compilation and execution in Java-enabled mobile devices, In Proc. *17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.
20. H. Saputra, R. R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Code protection for resource-constrained embedded devices," *LCTES '04 Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004, Accepted for publication, March 2004.
21. M. Pirretti, G. M. Link, R. R. Brooks, V. Narayanan, M. Kandemir, M. J. Irwin, "Fault tolerant algorithms for network-on-chip interconnect," ISVLSI 2004, Accepted for publication.
22. H. Saputra, R. R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Exploiting Value Locality for secure Energy Aware Communication," *IEEE Workshop on Signal Processing*, Seoul, Korea, August 2003.

National technology standards:

23. *J.M. Zachary, R. R. Brooks, and D. Thompson, "Secure Integration of Building Networks into the Global Internet,"* NIST GCR 02-837, National Institute of Standards and Technology, US Dept. of Commerce, Gaithersburg, MD, Oct. 2002.

Technical Reports:

24. R. R. Brooks, *CA Model of Mobile Code*, ARL Technical Memorandum, MUSE-TR-01.
25. J. Zachary, and R. R. Brooks, *Bi-directional Mobile Code Trust Management Using Tamper Resistant Hardware*, MUSE-TR-02.
26. R. R. Brooks and N. Gautam, *Analysis of Key Management using Peer-to-Peer Infrastructure,* MUSE-TR-03.
27. N. Vijaykrishnan, M. Kandemir, and R. R. Brooks, *Control flow Obfuscation of Java Codes, PSU CSE technical report.*
28. A. Kapur, N. Gautam, R. R. Brooks, and S. Rai, *"On Strategic-Level Design Optimization Problems in P2P Networks for Mobile Code Applications,"* MUSE-TR-04.
29. R. R. Brooks, S. A. Racunas, S. Rai, and N. Gautam, *"On Path Dependability in Random Graph Models,"* MUSE-TR-05.

Student theses:

- N. Orr, A Message Based Taxonomy of Mobile Code for Quantifying Network Communication, PSU CSE, Master's Thesis, Summer 2002.
- A. Kapur, Optimal Design of P2P Networks Supporting QoS Issues for Efficient File Sharing, PSU IE, Master's Thesis, Fall 2002.
- Eric Swankoski, Encryption and Security in Field-Programmable Gate Arrays, PSU CSE, Master's Thesis, Spring 2004.

9. LIST OF PERSONNEL ASSOCIATED

Dr. Richard R. Brooks - PI - Head, Distributed Systems Dept. Applied Research Laboratory of The Pennsylvania State University.

Dr. Shashi Phoha – Associate Director, Information Sciences and Technology Division, Applied Research Laboratory of The Pennsylvania State University.

Dr. Vijaykrishnan Narayanan- Associate Professor, Computer Science and Engineering, The Pennsylvania State University.

Dr. Mahmut Kandemir – Assistant Professor, Computer Science and Engineering, The Pennsylvania State University.

Dr. Natarajan Gautam – Associate Professor, Industrial and Manufacturing Engineering, The Pennsylvania State University.

Dr. John M. Zachary – Research Associate, Distributed Systems Dept. Applied Research Laboratory of The Pennsylvania State University. (Currently, Assistant Professor, Computer Science, University of South Carolina at Columbia.)

Mr. Eric Grele - Research Engineer. Applied Research Laboratory of The Pennsylvania State University.

Mr. Christopher Griffin – Research Engineer. Applied Research Laboratory of The Pennsylvania State University. Currently pursuing a masters degree in mathematics.

Mr. Art Jones - Research Engineer. Applied Research Laboratory of The Pennsylvania State University. Currently pursuing a Ph.D. in Information Science and Technology.

Mr. John Koch - Research Engineer. Applied Research Laboratory of The Pennsylvania State University.

Mr. Glenn Carl – Graduate Student. The Pennsylvania State University. Pursuing a Ph. D. in Electrical Engineering.

Mr. Nathan Orr – Graduate Student. Received an M. S. in Computer Science and Engineering from The Pennsylvania State University while working on the project.

Mr. Eric Swankoski – Graduate Student. Received an M. S. in Computer Science and Engineering from The Pennsylvania State University while working on the project.

Ms. Margaret Aichele - Undergraduate. The Pennsylvania State University. Computer Science and Engineering.

Mr. Thomas Keiser – Undergraduate. The Pennsylvania State University. Computer Science and Engineering.

Mr. Jason Schwier – Undergraduate. The Pennsylvania State University. Received a B. S. in Computer Science and Engineering from The Pennsylvania State University while working on the project.

Ms. Devaki Shah – Undergraduate. The Pennsylvania State University. Computer Science and Engineering.

Mr. Michael Young – Undergraduate. The Pennsylvania State University. Computer Science and Engineering.

10. PRESENTATIONS

The following papers were presented:

- H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the energy behavior of DES encryption, *The 6th Design Automation and Test in Europe Conference (DATE'03)*, Munich, Germany, March 2003. (Nominated for Best Paper)
- A. Kapur, N. Gautam, R. R. Brooks, and S. Rai, "Design, Performance and Dependability of a Peer-to-Peer Network Supporting QoS for Mobile Code Applications," *Tenth International Conference on telecommunications systems*, Sept. 2002.
- R. R. Brooks, C. Griffin, J. Zachary, and N. Orr, "An Interacting Automata Model for Network Protection," Invited Paper, *Fusion 2002*, July 2002.
- R. R. Brooks and C. Griffin, "Fugitive Search Strategy and Network Survivability," *2003 Industrial Engineering Research Conference*, invited Paper, January 2003.
- E. Swankoski, R.R. Brooks, V. Narayanan, M. Kandemir, and M. J. Irwin, "A Parallel Architecture for Secure FPGA Symmetric Encryption ," *Reconfigurable Architecture Workshop*, Santa Fe, New Mexico, April 2004.

- H. Saputra, N. Vijaykrishnan, M. Kandemir, R. Brooks, and M. J. Irwin. Exploiting value locality for secure energy aware communication. *2003 IEEE Workshop on Signal Processing Systems (SIPS'03)*, August 2003.
- G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R.Chandramouli. Energy-aware compilation and execution in Java-enabled mobile devices, *17th International Parallel and Distributed Processing Symposium (IPDPS'03),* April 2003.
- H. Saputra, R. R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Code protection for resource-constrained embedded devices," *LCTES '04 Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004.
- M. Pirretti, G. M. Link, R. R. Brooks, V. Narayanan, M. Kandemir, M. J. Irwin, "Fault tolerant algorithms for network-on-chip interconnect," ISVLSI 2004..
- H. Saputra, R. R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Exploiting Value Locality for secure Energy Aware Communication," *IEEE Workshop on Signal Processing*, Seoul, Korea, August 2003.

11. INVENTIONS

No new inventions have been declared. The CIP/URI efforts concentrated on performing basic research and creating proof-of-concept prototypes.

12. PROGRAM FINANCIAL SUMMARY

Total Project

	Actual	Budget
Salary		
Fringe Benefits		
Travel		
Equipment		
Tuition		
Overhead		
Col. Total	$0.00	$ -

13. REFERENCES

[Albert 2001] R. Albert and A.-L. Barabási, "Statistical Mechanics of Complex Networks," arXiv:cond-mat/0106096v1, June 2001.

[Barabzsi 1999]A-.L. Barabási and R. Albert, "Emergence of scaling in random networks," Science, vol. 286, pp. 509-512, 15 October 1999.

[Brooks 1998] R. R. Brooks and S. S. Iyengar, *Multi-sensor Fusion: Fundamentals and Applications with Software*, Prentice Hall PTR, Upper Saddle River, NJ, 1998.

[Brooks 2000] R. R. Brooks, E. Grele, W. Kliemkiwicz, J. Moore, C. Griffin, B. Kovak, and J. Koch "Reactive Sensor Networks: Mobile Code Support for Autonomous Sensor Networks," Distributed Autonomous Robotic Systems DARS 2000, Pp. 471-472. Springer Verlag, Tokyo, October 2000.

[Brooks 2002] R. R. Brooks, and N. Orr, "A Model for Mobile Code using Interacting Automata," *IEEE Transactions on Mobile Computing*, vol. 1, no. 4, pp. 313-326, October-December 2002.

[Camazine 2001] S. Camazine, et al, *Self-Organization in Biological Systems*, Princeton University Press, Princeton, N.J., 2001.

[CC04] Li-Chiou Chen and Kathleen M. Carley. The impact of countermeasure propagation on the prevalence of computer viruses. IEEE Transactions on Systems, Man and Cybernetics-Part B, 34(2):823– 833, April 2004.

[Chander 2001] A. Chander, J. Mitchell, I. Shin: Mobile code security by Java bytecode instrumentation. *DISCEX II*, 2001

[Cvetovic 1979] D. M. Cvetkovic, M. Doob, and H. Sachs, Spectra of Graphs, Academic Press, NY, 1979.

[Fuggetta 1998]A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342-361, May 1998.

[Grossglauer 1999]M. Grossglauer and J.-C. Blot, "On the Relevance of Long-Range Dependence in Network Traffic," IEEE/ACM Transactions on Networking, vol. 7, no. 5, pp. 629-640, October 1999.

[Howard 1998] J. D. Howard, T. A. Longstaff, *A Common Language for Computer Security Incidents*, Sandia Report, SAND98-8867.

[Howitz 1990] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (January 1990), 26-60.

[Jarvinen 2003] K. Jarvinen, M. Tommiska, and J. Skytta, "A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor," Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays, Pages 207-215.

[Kapur 2002] A. Kapur, N. Gautam, R. R. Brooks, and S. Rai, "Design, Performance and Dependability of a Peer-to-Peer Network Supporting QoS for Mobile Code Applications," Tenth International Conference on telecommunications systems, Sept. 2002.

[Krishnamachari 2001] Bhaskar Krishnamachari, Stephen B. Wicker, and Ramon Bejar, "Phase Transition Phenomena in Wireless Ad-Hoc Networks," Symposium on Ad-Hoc Wireless Networks, GlobeCom2001, San Antonio, Texas, November 2001. http://www.krishnamachari.net/papers/phaseTransitionWirelessNetworks.pdf

[Leland 1994] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)" IEEE/ACM Transactions on Networking, vol. 2, no. 1, pp. 1-15, February 1994..

[Milojicic 1999] D. Milojicic, F. Douglis, and R. Wheeler, ed.s, *Mobility: Processes Computers, and Agents*, Addison-Wesley, Reading, MA, 1999.

[Necula 1996] G.C. Necula, P. Lee: Safe kernel extensions without run-time checking. *In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[Oram 2001] A. Oram, ed. *"Peer-to-Peer Harnessing the Power of Disruptive Technologies"*, O'Reilly, Beijing, 2001.

[Orr 2002] N. Orr, A Message-Based Taxonomy of Mobile Code for Quantifying Network Communication, M. S. Thesis, Computer Science and Engineering, The Pennsylvania State University, Summer 2002.

[Portugali 2000] J. Portugali, *Self-Organization and the City*, Springer Series in Synergetics, Springer Verlag, Berlin, 2000.

[Rubin 1998] A. D. Rubin, and D. E. Geer, "Mobile Code Security," *IEEE Internet Computing*, pp. 30-34, Nov-Dec 1998.

[Saputra 2004] H. Saputra, R. R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Code protection for resource-constrained embedded devices," *LCTES '04 Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004, Accepted for publication, March 2004.

[Stallings 1995] W. Stallings, Network and Internetwork Security, Prentice Hall, Upper Saddle River, NJ, 1995.

[Tanenbaum 1997] A. S. Tananbaum and A. S. Woodhull, Operating Systems: Design and Implementation, Prentice Hall, Upper Saddle River, NJ, 1997.

[Watts 1999] D. J. Watts, *"Small Worlds"*, Princeton University Press, Princeton, NJ, 1999.

[Willinger 1998] W. Willinger and V. Paxson, "Where Mathematics Meets the Internet," *Notices of the AMS*, pp. 961-971, September 1998.